Universitat Jaume I de Castellón
E. S. de Tecnología y Ciencias Experimentales

Utilización del Paralelismo Multihebra en el Precondicionado y la Resolución Iterativa de Sistemas Lineales Dispersos

Castellón, Julio de 2010

Tesis Doctoral Presentada por: Alberto Fco. Martín Huertas
Dirigida por: José I. Aliaga Estellés
Enrique S. Quintana Ortí

Universitat Jaume I de Castellón
E. S. de Tecnología y Ciencias Experimentales

Utilización del Paralelismo Multihebra en el Precondicionado y la Resolución Iterativa de Sistemas Lineales Dispersos

Alberto Fco. Martín Huertas

Agradecimientos

Deseo expresar mi más sincero agradecimiento:

A mis directores, los doctores José I. Aliaga y Enrique S. Quintana, por su alto grado de diligencia, convicción, rigurosidad, apoyo científico e implicación durante la elaboración de este trabajo.

Al profesor doctor Matthias Bollhöfer de la Universidad Técnica de Braunschweig en Alemania, por su permanente disposición, generosidad y valiosas sugerencias.

A la Generalidad Valenciana, por su implicación en la investigación y soporte económico a través de las ayudas para la formación de personal investigador BFPI/06.

A mis compañeros del grupo de investigación, departamento y universidad, por su continuo y afectuoso aliento.

A mi amigo Ramón Feenstra, por aprender a escucharme en los momentos de incertidumbre, frustración y dudas.

A mis padres, Alfonso y Mª del Rosario, y hermano, Alfonso, por brindarme un hogar cálido y proporcionarme una educación en valores.

A Mª Jesús, en especial, por su amor y comprensión.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Planteamiento del problema

Los avances drásticos en el *software* y *hardware* de los computadores durante las últimas décadas han posibilitado el desarrollo de un enfoque moderno para la investigación científica, conocido indistintamente como ciencia computacional (*computational science*), computación científica (*scientific computing*), o modelado y simulación (*modeling and simulation*). La computación científica es un enfoque *multidisciplinar* para el análisis y resolución de problemas complejos de ciencia e ingeniería, mediante la *integración* de principios, técnicas y herramientas propias de ciencias como la astrofísica, la mecánica cuántica, la química, la biología molecular, la hidrodinámica o el electromagnetismo, de las *matemáticas aplicadas* y de las ciencias de la *computación*. Es por ello que el ejercicio de la computación científica requiere una compresión exhaustiva del problema, y destrezas avanzadas para el modelado matemático, análisis numérico, desarrollo de algoritmos, implementación de software, ejecución de programas, análisis, validación y visualización de los resultados computacionales.

El resultado final de la ciencia computacional es un *modelo computacional* que, mediante simulación numérica, permite estudiar sistemas complejos y fenómenos naturales que resultaría demasiado caro, peligroso, o incluso imposible estudiar mediante experimentación directa. De hecho, en muchas aplicaciones de ciencia e ingeniería ya se ha cruzado la frontera a partir de la cual la simulación (o la combinación apropiada de simulación y experimentación) es más efectiva (en términos de tiempo, coste y/o precisión) que la experimentación. Algunos problemas en los que se ha utilizado intensivamente la computación científica incluyen la predicción meteorológica y climatológica, la simulación, diseño y control de vehículos, el diseño de aviones, y el diseño electrónico. Asimismo, la computación científica es clave para el análisis de los datos experimentales obtenidos en el laboratorio en áreas científicas como la astrofísica, mecánica cuántica, física y biología molecular, así como para el desarrollo de nuevos modelos en campos emergentes como la biomedicina, la química computacional, la ciencia de materiales o la bioingeniería; en [149] y en sus referencias puede encontrarse una descripción más detallada de estas aplicaciones.

La búsqueda de mayor nivel de detalle, precisión y realismo en las simulaciones requiere típicamente una enorme capacidad computacional, y ha supuesto un impulso para la consecución de importantes avances en los algoritmos y la arquitectura de los computadores. Estos avances se enmarcan dentro de una disciplina conocida como "supercomputación" (*supercomputing*) o computación de altas prestaciones (*high performance computing*), que se encarga del análisis, diseño, y desarrollo de:

1. Arquitecturas de alto rendimiento.

2. Principios, técnicas y herramientas para la programación eficiente de estas arquitecturas.

Gracias a la supercomputación, los científicos e ingenieros pueden resolver actualmente problemas de gran dimensión considerados intratables en el pasado.

La *computación paralela* [79] resulta *cada vez más* un recurso efectivo y viable de la supercomputación en términos de coste/rendimiento. El desarrollo de aplicaciones paralelas se ha considerado *tradicionalmente* como una tarea costosa en tiempo y esfuerzo, principalmente por la complejidad que supone la especificación y coordinación de tareas concurrentes, y la falta de algoritmos portables, entornos estandarizados y herramientas para el desarrollo de aplicaciones. Además, si se analiza el *rápido crecimiento* de la capacidad computacional de los microprocesadores durante las décadas de los 70, 80, 90 y comienzos de la década del 2000, podría cuestionarse la necesidad de dedicar tanto esfuerzo al desarrollo de algoritmos y programas paralelos como recurso para acelerar las aplicaciones. En 1965, Gordon Moore predijo que la capacidad de integración en los circuitos (número de transistores por circuito integrado) se doblaría aproximadamente cada 18 meses. Esta predicción, conocida como "Ley de Moore", se ha cumplido de forma sorprendente a lo largo de los años. Sin embargo, la forma en la que este crecimiento en el número de transistores se aprovecha para conseguir un aumento de la capacidad computacional del microprocesador ha sido, es y será fruto de intenso debate y un desafío *arquitectural* clave. Esto es debido a la existencia de factores ineludibles en el diseño e implementación (tecnología) del hardware, que merman el crecimiento de la capacidad computacional de las arquitecturas monoprocesador, como el limitado grado de paralelismo a nivel de instrucción de los códigos reales, la latencia de acceso a memoria, y el consumo de energía y la disipación de calor de la tecnología actual. Un recurso lógico y clave para gestionar estas limitaciones es el uso *explícito* del paralelismo en el hardware, por ejemplo, en forma de múltiples núcleos de procesamiento dentro de un mismo microprocesador (procesadores multinúcleo o *multicore*). Estos diseños explícitamente paralelos han venido acompañados, en las dos últimas décadas, por un progreso considerable en la estandarización de los lenguajes y entornos de programación paralelos, asegurando un ciclo de vida más largo para el software.

La resolución de sistemas de ecuaciones lineales *dispersos* y de *gran dimensión* está dentro de los problemas matriciales más importantes y que surge con mayor frecuencia en el marco de la computación científica [56, 138]. Un sistema de ecuaciones lineales es disperso y de gran dimensión si su matriz de coeficientes presenta un gran número de entradas nulas y, además, su número de ecuaciones es $O(10^5)$, $O(10^6)$ o superior. Este tipo de problemas surge en áreas tan variadas como, por ejemplo, la acústica, el control del tráfico aéreo, la astrofísica, la bioquímica, la ingeniería química, la cinética química, la economía, la ingeniería eléctrica, la hidrodinámica, el análisis de estructuras, el electromagnetismo, la ingeniería del petróleo o el modelado de reactores nucleares. En estas aplicaciones, la resolución de los sistemas es típicamente una de las tareas computacionalmente más costosas y, por tanto, la aceleración del proceso de resolución se traduce directamente en una reducción del tiempo de simulación.

1.2. Resolución de sistemas de ecuaciones lineales dispersos

En multitud de problemas de ciencia e ingeniería surgen típicamente matrices con un gran número de entradas nulas, denominadas *matrices dispersas*. El interés de este tipo de matrices radica en el enorme *ahorro* computacional (consumo de memoria y número de operaciones) que se puede lograr aprovechando su estructura (y propiedades) mediante algoritmos y estructuras de datos adaptadas. A pesar de su ubicuidad, no existe una caracterización formal de las matrices dispersas. En general, una matriz es dispersa si resulta ventajoso computacionalmente explotar sus entradas nulas (con respecto a no hacerlo). Una caracterización algo más formal, que se utiliza comúnmente, establece que una matriz $A \in \mathbb{R}^{n,n}$ es dispersa si

$$nnz(A) = O(n),$$

donde $nnz(A)$ es el número de entradas no nulas de A. Por otro lado, existe otra regla que caracteriza A como dispersa si existe un número $p \ll n$, tal que toda fila y columna de A tiene a lo sumo p entradas no nulas. Un término que también está conectado con la noción de dispersidad

es la *densidad* de una matriz, que se define como el cociente $nnz(A)/n^2$. El *patrón* de elementos no nulos de una matriz A, se define como el conjunto $\mathcal{A} = \{(i,j) : a_{ij} \neq 0\} \subseteq \{1,\ldots,n\}\times\{1,\ldots,n\}$ y, por tanto, $nnz(A)$ coincide con la cardinalidad del conjunto $\mathcal{A}$, $|\mathcal{A}|$. Una matriz es *densa* si no tiene o tiene "muy pocas" entradas nulas. En esta tesis, utilizaremos las expresiones "más dispersa" o "más densa" para referirnos a una matriz que tiene menos o más entradas no nulas que otra, respectivamente.

La resolución de sistemas de ecuaciones lineales *dispersos* y de *gran dimensión* consiste en encontrar la solución $x \in \mathbb{R}^n$ al sistema de ecuaciones lineales (expresado en forma matricial como)

$$Ax = b, \tag{1.1}$$

donde $A \in \mathbb{R}^{n,n}$ es una matriz dispersa, de gran dimensión y de estructura arbitraria, y $b \in \mathbb{R}^n$ es el vector de términos independientes. La fuente fundamental de problemas matriciales dispersos y de gran dimensión es la discretización de ecuaciones diferenciales en derivadas parciales (EDPs) [138]. Sin embargo, este tipo de sistemas lineales también surge en aplicaciones no gobernadas por EDPs, como el análisis, diseño y simulación de circuitos por computador [138].

Tradicionalmente, los métodos numéricos para la resolución de (1.1) se han divido en directos o iterativos. Aunque emplearemos esta clasificación para definir el marco de la tesis, es necesario puntualizar que se trata de una simplificación extrema y que, por tanto, puede llevar a confusión si se utiliza como referencia para describir el estado actual de los métodos numéricos de resolución de sistemas de ecuaciones lineales dispersos. Por una parte, los límites entre ambas categorías son cada vez más difusos, principalmente por el considerable abanico de ideas y técnicas que están siendo transferidas desde el área de los métodos directos al campo de los métodos iterativos (habitualmente en forma de *precondicionadores*), mejorando la fiabilidad, robustez y eficiencia de éstos últimos. Por otra, mientras que la mayoría de los métodos directos están basados en el proceso de eliminación Gaussiana (o en alguna variante de la misma), el campo de los métodos iterativos engloba una variedad *abrumadora* de técnicas de *precondicionado* y métodos de resolución, que abarca los métodos iterativos de punto fijo, en los que la convergencia no está garantizada para todo tipo de sistemas (como por ejemplo, el método SOR [138]), los métodos basados en la generación de subespacios de Krylov, que teóricamente convergen en número finito de pasos en aritmética exacta (como por ejemplo, el método GMRES [139] o el CG [93]), o los métodos multinivel, que presentan escalabilidad algorítmica temporal (número de operaciones) óptima para una gran variedad de aplicaciones gobernadas por EDPs (como por ejemplo, los métodos *multigrid* algebraicos y geométricos [116, 156]). Incluso algunos de los métodos que originalmente fueron diseñados para resolver el sistema (1.1), pueden tener un papel relevante cuando se utilizan como *precondicionadores* y, por tanto, como bloques integrantes de resolutores más sofisticados.

En la década de los 60, 70 y 80, los métodos directos [56] de resolución eran con frecuencia la alternativa elegida para la resolución de grandes sistemas, debido a su robustez y comportamiento predecible. Sin embargo, en las últimas décadas los métodos iterativos [138] han ganado gradualmente terreno a los métodos directos de resolución por los siguientes factores. En primer lugar, el tamaño y complejidad de la nueva generación de sistemas que surge en las aplicaciones más comunes ha crecido significativamente, hasta tal punto que la resolución de modelos tridimensionales pone en serios apuros a los métodos directos de resolución más eficientes en la actualidad, debido a su elevado volumen de cálculo y consumo de memoria. Además, la robustez, fiabilidad y eficiencia de los métodos iterativos ha mejorado significativamente gracias a los avances en técnicas de *precondicionado*. En particular, estas técnicas constituyen el ingrediente más crítico a la hora de mejorar el rendimiento de este tipo de métodos. Por último, es significativamente más sencillo desarrollar métodos iterativos paralelos que versiones paralelas de los métodos directos.

1.2.1. Resolución directa

La mayoría de los métodos directos empleados en la práctica para resolver sistemas de ecuaciones lineales se basan en un proceso conocido como factorización LU o eliminación Gaussiana [76, 164]. Este proceso descompone la matriz de coeficientes del sistema (1.1), A, en dos matrices triangulares, $L \in \mathbb{R}^{n,n}$ triangular inferior unidad y $U \in \mathbb{R}^{n,n}$ triangular superior, de forma que $A = LU$. Posteriormente, la solución de (1.1) se puede obtener mediante la resolución de dos sistemas lineales con estructura triangular, empleando los factores triangulares como matriz de coeficientes; en particular, resolviendo en primer lugar $Ly = b$ y, posteriormente, $Ux = y$.

Los métodos directos para matrices *dispersas* tratan de reducir el coste computacional y de almacenamiento de la eliminación Gaussiana evitando operar con las entradas nulas y minimizando el *llenado* que se produce en el proceso de factorización [56, 62]. La algorítmica de estos métodos se basa en el uso de reordenaciones para reducir el llenado y en el particionado del conjunto de incógnitas del sistema para identificar bloques densos en los factores. La estructura por bloques de los factores habilita la utilización de rutinas básicas de álgebra lineal de tercer nivel (BLAS3 [58]), cuyo uso es imprescindible para obtener altas prestaciones en los procesadores superescalares modernos. Gran parte del éxito de los métodos resultantes depende de su capacidad para gestionar de forma eficiente el llenado. Aunque los métodos directos actuales son capaces de resolver sistemas de ecuaciones moderadamente grandes, su coste espacial y temporal escala pobremente con el tamaño del problema, especialmente para matrices que surgen de la discretización de EDPs tridimensionales, a causa del llenado masivo que se produce en el proceso de factorización. En cuanto a sus ventajas, cabe indicar que son métodos extremadamente fiables y numéricamente robustos y, además, con frecuencia se pueden predecir los recursos computacionales y de almacenamiento que requieren. Es por ello que suelen ser los elegidos para resolver sistemas de ecuaciones en entornos industriales, en los que la fiabilidad es clave. Tradicionalmente se han empleado en las áreas de análisis de estructuras, modelado de dispositivos semiconductores, dinámica de fluidos computacional, y prácticamente en todas las aplicaciones no gobernadas por EDPs.

1.2.2. Resolución iterativa

En sistemas de ecuaciones lineales de dimensión extremadamente grande y/o en los que se produce un elevado nivel de llenado al aplicar métodos directos, los métodos iterativos son la única alternativa viable. Esta familia de métodos resuelve el sistema a través de una secuencia de transformaciones que se aplican a una solución inicial de partida, y que aproximan progresivamente la solución. Estas transformaciones se agrupan en etapas o iteraciones, de tal forma que los resultados correspondientes a las transformaciones que se efectúan en una iteración constituyen la entrada para la siguiente. Esta propiedad los hace especialmente eficientes para resolver sistemas de ecuaciones lineales de gran dimensión, ya que el proceso puede detenerse cuando se logra la precisión deseada. Aunque los métodos iterativos requieren menos memoria, y habitualmente efectúan menos operaciones aritméticas que los métodos directos (especialmente cuando la aplicación no requiere una solución de alta precisión), son métodos menos fiables. Para algunas aplicaciones, los métodos iterativos fallan y, además, habitualmente se hace necesaria la aplicación de técnicas de *precondicionado* para acelerar su convergencia. A pesar de que los métodos iterativos de resolución han madurado significativamente, todavía hay muchos problemas por resolver. En particular, no se puede asegurar que un sistema de ecuaciones arbitrario pueda ser resuelto por un método iterativo de forma eficiente. Tradicionalmente los métodos iterativos se han empleado en la industria del petróleo, y en la de la energía nuclear, aunque su rango de aplicación se ha ampliado considerablemente como fruto de la investigación efectuada en los últimos años en técnicas de precondicionado [138].

Seguidamente se presentan los fundamentos básicos de los métodos iterativos más populares, y el objetivo principal de las técnicas de precondicionado.

Fundamentos básicos de los métodos iterativos

El objetivo de este subapartado es introducir los fundamentos básicos de los métodos iterativos más populares, y revisar su evolución histórica. Estos métodos se pueden clasificar considerando dos periodos de tiempo caracterizados por la familia/tipo de métodos en los que se concentraron principalmente los esfuerzos de investigación. El primer periodo comienza aproximadamente en los años 50, finaliza a principios de los 70, y está dominado por los métodos iterativos *estacionarios*; el segundo comienza a mediados de los 70 y en éste predominan los métodos iterativos *basados en la generación de subespacios de Krylov*. El desarrollo de los métodos *multigrid* geométricos y algebraicos también pertenece a este segundo periodo.

El comienzo del primer periodo queda simbolizado por el trabajo realizado por Young en su tesis doctoral [168]. En esta época, los métodos iterativos estacionarios, como por ejemplo, los métodos de Jacobi, Gauss-Seidel, SOR y SSOR [138], se aplicaban ampliamente para la resolución de sistemas de ecuaciones surgidos de la discretización de EDPs del tipo elíptico. Matemáticamente, los métodos iterativos estacionarios se pueden describir a través de la recurrencia

$$x^{(k+1)} \leftarrow Tx^{(k)} + c, \ \textbf{con}\ k = 0, 1, 2, \ldots$$

donde $x^{(k)}$ denota la aproximación a la solución de $Ax = b$ en la iteración k-ésima, T es una matriz de iteración predefinida (fija) que caracteriza el método particular, c es un vector predefinido (fijo) y $x^{(0)}$ es una aproximación inicial. En este periodo, también se consideró el uso de técnicas de aceleración de Chebyshev para aumentar la velocidad de convergencia de métodos iterativos como el SSOR, y se desarrolló el método ADI (*Alternating Direction Implicit*) [60, 127], un serio competidor del método SOR. Un evento crucial para este periodo es la publicación del libro de Varga [163], que contiene los resultados de un exhaustivo análisis matemático de los métodos iterativos estacionarios. El final de este periodo se puede datar en la publicación del libro de Young [169], que contiene el tratamiento analítico definitivo de los métodos SOR, ADI y otros métodos iterativos clásicos.

A pesar de su elegancia matemática, los métodos iterativos estacionarios tienen *serias* limitaciones, como por ejemplo, su falta de aplicabilidad o la dependencia de parámetros para los cuales es difícil encontrar un valor apropiado sin manejar información específica sobre el espectro de valores propios de A. Como resultado, este tipo de métodos pueden llegar a converger muy lentamente, o incluso no llegan a converger para muchos problemas de interés práctico. El texto de Hageman y Young [87] cubre procedimientos para estimar de forma adaptativa el valor de estos parámetros, junto con el uso de técnicas de aceleración basadas en la generación de subespacios de Krylov (por aquel entonces emergentes). En cierto sentido, este libro marca la transición del primer al segundo periodo en la historia de los métodos iterativos.

El segundo periodo está dominado por los métodos basados en la generación de subespacios de Krylov. En las últimas décadas han surgido multitud de textos [17, 22, 41, 43, 70, 80, 86, 119, 138, 166] dedicados por completo a los métodos iterativos para la resolución de sistemas de ecuaciones lineales. Estas obras son la culminación de años de investigación sobre el desarrollo de nuevos métodos, particularmente basados en la generación de subespacios de Krylov. Matemáticamente, todos los métodos que pertenecen a esta familia extraen, en cada iteración $k = 1, 2, \ldots$, una aproximación $x^{(k)}$ a la solución de $Ax = b$ dentro del subespacio

$$x^{(0)} + \mathcal{K}_k(A, r^{(0)}) = x^{(0)} + \text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}\},$$

donde $x^{(0)}$ es una solución de partida, $\mathcal{K}_k(A, r^{(0)})$ es el subespacio de Krylov de dimensión k, generado por la matriz A y el residuo inicial $r^{(0)} = b - Ax^{(0)}$, y $\text{span}\{r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}\}$ es el subespacio generado por los vectores $r^{(0)}, Ar^{(0)}, A^2r^{(0)}, \ldots, A^{k-1}r^{(0)}$. Los distintos métodos se diferencian entre sí por las restricciones que imponen para extraer $x^{(k)}$ de este subespacio, pudiendo distinguir principalmente entre los cuatro enfoques que enumeramos a continuación:

1. El enfoque de *Ritz-Galerkin* escoge $x^{(k)}$ de modo que el residuo $r^{(k)} = b - Ax^{(k)}$ sea ortogonal al subespacio $\mathcal{K}_k(A, r^{(0)})$, constituyendo la base de métodos muy populares como el método CG, el método de Lanczos, y el método FOM y sus variantes IOM y DIOM.

2. El enfoque del *mínimo residuo* calcula $x^{(k)}$ para que $r^{(k)} = b - Ax^{(k)}$ sea ortogonal al subespacio $A \cdot \mathcal{K}_k(A, r^{(0)})$ o, equivalentemente, de tal forma que $\|b - Ax^{(k)}\|_2$ sea mínimo entre todos los vectores del subespacio $\mathcal{K}_k(A, r^{(0)})$. Los métodos GMRES, MINRES y ORTHODIR se basan en esta aproximación.

3. El enfoque de *Petrov-Galerkin* construye $x^{(k)}$ de forma que $r^{(k)} = b - Ax^{(k)}$ sea ortogonal a otro subespacio apropiado de dimensión k, $\mathcal{L}_k$. Cuando $\mathcal{L}_k = \mathcal{K}_k(A^T, r^{(0)})$ se obtienen métodos como el Bi-CG y el QMR.

4. El enfoque del *mínimo error* determina $x^{(k)}$ imponiendo que $\|x - x^{(k)}\|_2$ sea mínimo entre todos los vectores del subespacio $A^T \cdot \mathcal{K}_k(A^T, r^{(0)})$. Los métodos SYMMLQ y GMERR pertenecen a esta clase.

Es necesario mencionar que existen otros métodos como el CGS, Bi-CGSTAB, FGMRES y GMRESR que combinan ideas de estos enfoques; en [17, 22, 41, 43, 70, 80, 86, 119, 138, 166] pueden encontrarse más detalles sobre los fundamentos, análisis matemático y numérico de los métodos basados en la generación de subespacios de Krylov. El resto de este apartado describe los hitos más importantes en los desarrollos llevados a cabo durante el segundo periodo.

En 1952, Lanczos [108] y Hestenes y Stiefel [93] descubrieron (de forma independiente y casi simultáneamente) el método de los gradientes conjugados (*Conjugate Gradients*, CG) para la resolución de sistemas de ecuaciones lineales en los que A es simétrica y definida positiva (SDP). Este método se clasificó originalmente como directo, ya que teóricamente su convergencia está garantizada en no más de n iteraciones. De forma más precisa, asumiendo aritmética exacta, el método converge a la solución exacta en m pasos, donde m es el número de valores propios distintos de la matriz A. Mediante experimentación numérica se observó que, en presencia de errores de redondeo, el método requiere algo más de n iteraciones para obtener una aproximación precisa, especialmente para sistemas mal condicionados. Por otra parte, en [93] se observó que para sistemas bien condicionados, se obtenían aproximaciones satisfactorias en muchas menos de n iteraciones, constituyendo este método una alternativa atractiva a la eliminación Gaussiana. Sin embargo, por las razones aducidas en [77], el método CG se utilizó relativamente poco en los 20 años posteriores a su definición, años durante los cuales la eliminación Gaussiana para matrices densas, y el método SOR y la iteración de Chebyshev para dispersas, eran los métodos de resolución más empleados.

Esta situación cambió finalmente alrededor del año 1970, con la publicación de un artículo de Reid [133]. En este trabajo, Reid demostró que, para sistemas de ecuaciones lineales dispersos y de gran dimensión razonablemente bien condicionados, CG es un método iterativo muy potente, que permite obtener buenas aproximaciones a la solución en muchas menos de n iteraciones. Este trabajo supuso el inicio de la rehabilitación del método CG y estimuló la investigación en varias direcciones; por una parte, la extensión del método CG a sistemas de ecuaciones indefinidos y/o simétricos; por otra, el desarrollo de técnicas de precondicionado para mejorar el condicionamiento de la matriz y aumentar de esa forma la velocidad de convergencia del método [54].

La extensión del método CG a sistemas simétricos indefinidos, realizada por Paige y Saunders, dio lugar a los métodos MINRES y SYMMLQ [125]. A finales de los 70 y comienzos de los 80, varios investigadores se centraron en el caso no simétrico. Este trabajo, que continuó a lo largo de los 80, culminó con el desarrollo del método GMRES por Saad y Schultz [139], el método QMR por Freund y Nachtigal [72], y el método Bi-CGSTAB de van der Vorst [159]. En los años noventa, la investigación se centró principalmente en el análisis y mejora de las técnicas existentes más que en el desarrollo de nuevos métodos. Un avance importante en esta década

fue el análisis completo del efecto de los errores de redondeo sobre varios métodos iterativos basados en la generación de subespacios de Krylov [61, 81]. También es importante mencionar la aparición de variantes flexibles [78, 122, 136, 155] de este tipo de métodos, que permiten que el precondicionador cambie de una iteración a otra. Esta técnica posibilita, por ejemplo, que un método iterativo cualquiera pueda actuar como precondicionador de un método flexible.

Otro evento crucial en el ámbito de los métodos iterativos fue la formulación de los métodos multimalla (*multigrid*) de Brandt [36] y Hackbusch [85]. Los métodos multimalla pertenecen a la familia de los métodos multinivel, que están especialmente diseñados para resolver de forma rápida y eficiente sistemas de ecuaciones lineales que surgen de la discretización de EDPs. Estos métodos se pueden emplear de forma aislada, es decir, como métodos iterativos, o como precondicionadores del método CG u otros métodos basados en la generación de subespacios de Krylov. Los métodos multimalla giran entorno a un método estacionario, como por ejemplo, el método de Jacobi o Gauss-Seidel. Estos últimos reducen muy lentamente la magnitud de las componentes de baja frecuencia del error, aunque son muy rápidos para reducir las componentes de alta frecuencia. Los métodos multimalla explotan la posibilidad que se da en este tipo de problemas de construir un sistema más "grueso" (*coarser*), y transformar las componentes del error de baja frecuencia del sistema original en componentes de alta frecuencia en el sistema más grueso, para de esa forma poder reducir su magnitud usando un método iterativo estacionario. Si se aplica esta estrategia recursivamente a través de una jerarquía de sistemas (mallas o niveles en la terminología multimalla), se obtiene un método cuyo coste computacional (número de operaciones) depende sólo linealmente del tamaño del problema. Esta propiedad de los métodos multimalla se denomina escalabilidad algorítmica. Los métodos multimalla originales, también denominados geométricos, eran métodos específicos para EDPs elípticas de segundo orden en mallas estructuradas y geometría muy simple. Con el tiempo, el ámbito de los métodos multimalla, y de forma más general, el de los métodos multinivel, se ha ampliado progresivamente, incluyendo clases de EDPs, discretizaciones y geometrías cada vez más sofisticadas. Un avance importante en este sentido fue la introducción de los métodos multimalla algebraicos (*Algebraic MultiGrid*, AMG) en los años 80 [37, 38, 39, 134]; este tipo de métodos sólo utilizan la información contenida en A y, por tanto, son más apropiados para resolver EDPs en dominios complicados y mallas irregulares. Desde entonces se han realizado muchos esfuerzos de investigación en este tipo de métodos y se han desarrollado muchas variantes nuevas, como por ejemplo, *smoothed aggregration* [160, 161], AMGe [40] y *spectral* AMG [46], por nombrar algunas. También se han elaborado varios textos [116, 156] sobre esta familia de métodos.

La escalabilidad algorítmica de los métodos multinivel es clave para desarrollar algoritmos paralelos que mantengan su eficiencia a medida que crece el tamaño del problema en la misma proporción que el número de procesadores. Además, la jerarquía intrínseca de los métodos multinivel permite adaptarlos a la jerarquía de memoria así como a los tipos de paralelismo disponibles en las plataformas de computación de altas prestaciones actuales, a través de una combinación apropiada del número de niveles y del tamaño de los problemas a resolver en cada uno de estos niveles. Estas propiedades favorables han hecho que los métodos AMG se conviertan en el ingrediente principal de los paquetes de software de métodos iterativos paralelos actuales; una revisión de los enfoques utilizados para la paralelización del métodos AMG puede encontrarse en [167] y sus referencias.

Otra familia importante de esquemas iterativos es la de los métodos de descomposición de dominios [132, 150], que se popularizaron en los años 80 principalmente por la aparición de plataformas de computación masivamente paralelas. En la actualidad, los métodos de descomposición de dominios (y sus variantes multinivel) se utilizan de forma exclusiva como precondicionadores. Matemáticamente, estos métodos pueden considerarse como una extensión de los precondicionadores simples por bloques como los de Jacobi o Gauss-Seidel por bloques [138]. Más que una clase específica de algoritmos, los métodos de descomposición de dominios pueden ser considerados como un paradigma general para descomponer un problema en otros más pequeños con el

propósito de aplicar procesamiento paralelo.

Fundamentos básicos de las técnicas de precondicionado

Los métodos iterativos convergen rápidamente si la matriz del sistema es una buena aproximación de la matriz identidad, a la que nos referiremos con la letra I. En la mayoría de las aplicaciones la matriz A no cumple esta propiedad; sin embargo, es posible reemplazar (1.1) por alguno de los dos siguientes sistemas de ecuaciones lineales equivalentes

$$M^{-1}Ax = M^{-1}b \quad \text{o} \tag{1.2}$$

$$AM^{-1}\hat{x} = b,\ x = M^{-1}\hat{x}, \tag{1.3}$$

con el objetivo de que $M^{-1}A$ o AM^{-1} sean buenas aproximaciones de la matriz identidad. En estos sistemas, $M \in \mathbb{R}^{n,n}$ es una matriz no singular a la que se llama *precondicionador*, y los propios sistemas (1.2) y (1.3) se conocen, respectivamente, como sistema precondicionado por la izquierda y sistema precondicionado por la derecha. Si A es una matriz simétrica ($A = A^T$) y M una matriz SDP ($M = M^T$ y $x^T Mx > 0\ \forall\ x \neq 0 \in \mathbb{R}^n$), entonces es posible construir el sistema precondicionado de tal forma que se preserve la simetría de la matriz de coeficientes. En concreto, si tenemos en cuenta que el precondicionador se puede factorizar como $M = LL^T$, con $L \in \mathbb{R}^{n,n}$ el factor de Cholesky de M [76], entonces en el sistema precondicionado

$$L^{-1}AL^{-T}\hat{x} = L^{-1}b, \tag{1.4}$$

la matriz de coeficientes $L^{-1}AL^{-T}$ es simétrica y $x = L^{-T}\hat{x}$. Si además de simétrica, A es definida positiva, entonces $L^{-1}AL^{-T}$ también lo es. Para resolver cualquiera de los sistemas precondicionados (1.2)-(1.4) mediante los métodos iterativos basados en la generación de subespacios de Krylov, no es necesario formar $M^{-1}A$, AM^{-1} o $L^{-1}AL^{-T}$ explícitamente. En su lugar, el requisito es poder resolver sistemas de ecuaciones lineales con M como matriz de coeficientes, es decir, sistemas de la forma $Mz = r$, donde r es conocido y z es la incógnita. En resumen, las técnicas de precondicionado son métodos numéricos que buscan un precondicionador M, con $M^{-1}A \approx I$ en "algún sentido", y coste relativamente moderado para la resolución de $Mz = r$.

El sentido exacto en el que el precondicionador debe hacer que $M^{-1}A \approx I$ depende del método iterativo escogido para resolver el sistema. En los métodos iterativos basados en la generación de subespacios de Krylov, este requisito se traduce con frecuencia en mejorar las propiedades espectrales de la matriz de coeficientes del sistema. En el caso simétrico, y para los métodos basados en criterios de optimalidad, como por ejemplo el CG o el MINRES, es posible derivar cotas precisas del error o del residuo (en función de sobre cuál de estas dos magnitudes se establezca el criterio de optimalidad) correspondiente a las aproximaciones que dependen exclusivamente del espectro de valores propios de la matriz. Además, debido a los términos en los que estas cotas están expresadas, es posible intuir lo que constituye un espectro favorable o desfavorable desde el punto de vista de la convergencia del método. En concreto, un espectro en el que la mayoría de los valores propios se agrupan entorno a un punto alejado del origen es favorable, mientras que un espectro es desfavorable si los valores propios están dispersos en un intervalo de gran longitud. En el caso no simétrico la situación es más compleja, y los valores propios no describen (exclusivamente) la convergencia de estos métodos iterativos, como por ejemplo en el método GMRES. Sin embargo, a nivel práctico, un espectro en el que la mayoría de los valores propios se agrupan entorno a un punto, habitualmente resulta en que el método converja rápidamente, especialmente cuando la matriz del sistema precondicionado es aproximadamente normal, es decir, cuando $AA^T \approx A^T A$.

En términos generales, las técnicas de precondicionado se pueden clasificar en dos categorías distintas. En primer lugar, las técnicas de *propósito específico* están diseñadas para un conjunto específico de aplicaciones, como por ejemplo, las gobernadas por EDPs de un tipo concreto. En muchas situaciones es posible detectar y usar información ligada a/derivada de la física del

problema para construir precondicionadores muy eficaces. Estas técnicas requieren un conocimiento completo de aspectos referentes al análisis, geometría y física de la aplicación subyacente incluyendo, por ejemplo, las ecuaciones continuas que gobiernan su comportamiento, el dominio de integración o detalles de la discretización. Ejemplos de esta categoría son las técnicas multimalla [116, 156] y las de descomposición de dominios [132, 150]. Sin embargo, este enfoque no siempre es deseable o posible. Habitualmente, estas técnicas suelen ser muy sensibles a variaciones en los detalles de la aplicación, e incluso cambios modestos pueden comprometer por completo la eficacia del método. Además, la información específica de la aplicación puede ser bastante compleja de usar u obtener. Por esta razón es útil disponer de métodos de *propósito general*, puramente *algebraicos*, que obtengan M a partir de la matriz del sistema original, sin necesidad de disponer de información específica sobre la aplicación de partida. Estas técnicas no proporcionan la solución óptima al problema, aunque ofrecen un rendimiento más que aceptable para un amplio rango de aplicaciones. Los precondicionadores algebraicos son más fáciles de desarrollar y de usar, por ejemplo, en aplicaciones que requieren discretizaciones que involucran mallas no estructuradas y, habitualmente, es posible mejorar su eficacia con ajustes específicos para cada problema. Dentro de esta categoría también se incluyen los precondicionadores destinados a una clase específica (aunque lo suficientemente general) de matrices, como por ejemplo, precondicionadores para matrices SDP o para M-matrices [138]. Las técnicas de precondicionado basadas en la factorización *aproximada* de A, o de su inversa A^{-1}, como por ejemplo, los precondicionadores ILU [138] (*Incomplete LU*) o los precondicionadores SPAINV [138] (*SParse Approximate Inverse*), respectivamente, son ejemplos de esta categoría de técnicas de precondicionado.

1.3. Arquitecturas paralelas y programación paralela

La incesante demanda de mayor realismo en las simulaciones motiva el desarrollo de sistemas de computación de altas prestaciones cada vez más potentes, y el desarrollo de arquitecturas más potentes posibilita a su vez la resolución de aplicaciones más costosas, con mayor nivel de detalle. La computación de altas prestaciones incluye en la actualidad una gran variedad de arquitecturas de alto rendimiento, desde procesadores superescalares y/o VLIW, pasando por procesadores multinúcleo de propósito general y aceleradores gráficos, hasta grandes servidores de cálculo conectados en red. En esta sección presentamos los principios básicos del funcionamiento de las arquitecturas de alto rendimiento en el marco de su evolución histórica (apartado 1.3.1), los estándares actuales para la programación de este tipo de arquitecturas (apartado 1.3.2), y una serie de bibliotecas actuales para la resolución iterativa de sistemas de ecuaciones lineales dispersos (apartado 1.3.3).

1.3.1. Arquitecturas paralelas

Aunque la era de la "Supercomputación" se inició con la introducción de los computadores vectoriales en los años 70, la aparición de los microprocesadores en la siguiente década, con un mejor ratio precio-rendimiento, barrió rápidamente del mercado general a los procesadores vectoriales y, posteriormente, también hizo desaparecer este tipo de arquitecturas del nicho de la computación de alto rendimiento [153, 154].

Desde entonces, y durante 20 años aproximadamente, el incremento en el número de transistores dictado por la Ley de Moore fue aprovechado por tecnólogos y expertos en arquitectura de computadores en un proceso constante de reducción del tamaño de los circuitos y diseño de una circuitería cada vez más sofisticada [90]. La consecuencia más visible de las mejoras en la escala de integración fue el aumento de la frecuencia de reloj de los circuitos integrados: mientras que el VAX 8700 de 1985 funcionaba a 12,5 MHz, 20 años más tarde, en 2005, el procesador Xeon de Intel alcanzaba los 3,6 GHz (un factor de mejora cercano a 290). Por otro lado, los avances

arquitectónicos se han plasmado principalmente en dos líneas: la incorporación de memorias caché dentro del chip, estructuradas en un mayor número de niveles y de mayor tamaño, para ocultar la latencia de acceso a la memoria principal (un procesador Itanium2 y las últimas versiones de los procesadores Intel Xeon incluyen hasta tres niveles de memoria caché dentro del chip, con unos tamaños que exceden, con mucho, los de la memoria principal de los microprocesadores de 20 años antes); y el diseño de potentes unidades de control capaces de explotar el paralelismo a nivel de instrucción (*instruction-level parallelism*, ILP) no sólo mediante la segmentación sino también mediante la emisión simultánea de múltiples instrucciones (procesadores superescalares), detectando y controlando las dependencias entre instrucciones, e incluso reordenando la secuencia de instrucciones del código para incrementar el grado de paralelismo.

Desde 2004, las limitaciones en consumo de energía y disipación de calor de la tecnología actual han desembocado en el diseño de procesadores con múltiples núcleos de procesamiento como principal medio para seguir aprovechando los avances en la escala de integración dictados por la Ley de Moore. El término genérico de *procesadores multinúcleo* engloba hoy en día dos aproximaciones bien diferentes: por un lado, los procesadores de propósito general, como los desarrollados por Intel y AMD; y, por otro, los aceleradores hardware como el Cell B.E. de IBM, las GPUs de NVIDIA y AMD/ATI, las FPGAs de Celoxica, etc. En la actualidad, los procesadores con múltiples núcleos superescalares dominan el mercado de la computación de propósito general, si bien distintos aceleradores hardware han demostrado un ratio precio-rendimiento mucho más elevado en ciertas aplicaciones, y pueden amenazar esta hegemonía en un futuro próximo. Se presenta aquí la paradoja de que, mientras que los aceleradores hardware intentan mejorar su versatilidad para parecerse más a un procesador de propósito general, algunos prototipos de procesadores de propósito general, como el Intel *Larrabee* [148], intentan copiar ideas propias de los aceleradores.

A continuación se ofrece una breve descripción de la organización y los principios de funcionamiento de estas arquitecturas, ligando estos aspectos con su evolución histórica.

Procesadores vectoriales

Una de las restricciones más serias para explotar el paralelismo a nivel de instrucción es la limitada disponibilidad de este tipo paralelismo en los programas. Sin embargo, algunos estudios han demostrado que las aplicaciones científicas, donde es habitual el tratamiento intensivo de vectores, tienen una gran cantidad de paralelismo a nivel de datos (*data-level parallelism*). Por eso algunas de las soluciones hardware que se presentan para explotar este paralelismo están orientadas al tratamiento de grandes estructuras de datos.

En particular, los *computadores vectoriales segmentados* son capaces de ejecutar instrucciones especiales para la manipulación de vectores. Para ello disponen de unidades funcionales segmentadas para procesamiento vectorial, capaces de dividir la ejecución de una instrucción vectorial que efectúa un cálculo sobre los elementos de un vector en múltiples etapas, solapando la ejecución de estas etapas (segmentación), y llegando a producir un resultado (es decir, un elemento procesado del vector) por ciclo de reloj. La clave del rendimiento de estos procesadores reside en que no existen dependencias de datos entre el procesamiento que debe efectuarse sobre los elementos de un vector. En otras palabras, los procesadores vectoriales extienden el principio de la segmentación para procesar los datos escalares contenidos en los vectores.

A pesar de que los principios que fundamentan el interés de los procesadores vectoriales siguen siendo válidos, las enormes ventajas en precio-rendimiento de la tecnología de microprocesadores llevó a las grandes compañías fabricantes de procesadores vectoriales (SG-Cray, Fujitsu, Hitachi, etc.) a sustituir este tipo de arquitectura por multiprocesadores de gran escala basados en microprocesadores. Los últimos vestigios de los computadores vectoriales son el SX-6 de NEC, y su versión mejorada, el SX-9 que constituye el elemento básico de construcción del supercomputador *Earth Simulator*.

Segmentación y procesadores superescalares

En la actualidad el mercado de procesadores de propósito general está dominado por los procesadores superescalares con ejecución fuera de orden. Este tipo de arquitecturas se caracteriza por realizar la captación de múltiples instrucciones por ciclo de reloj. En cada ciclo, varias de estas instrucciones son decodificadas y entran en una estructura donde se detectan las dependencias entre ellas. Aquellas que están libres de dependencias (es decir, que tienen sus operandos fuente disponibles), se emiten simultáneamente para su ejecución en las unidades funcionales. En este punto hay varias alternativas de implementación, desde las que proponen emitir las instrucciones a las unidades funcionales directamente, si éstas están disponibles, hasta las que emiten las instrucciones a unas colas (las estaciones de reserva) asociadas a las unidades funcionales, desde donde pasan a ejecutarse en orden cuando las propias unidades funcionales lo permiten. Una vez ejecutadas las instrucciones, éstas escriben sus resultados en el banco de registros y se retiran del cauce siguiendo el orden de programa para mantener la consistencia del programa.

En resumen, los procesadores superescalares combinan la ejecución solapada de instrucciones, propia de los procesadores segmentados, y añaden a ésta la posibilidad de ejecutar más de una instrucción en paralelo en la misma etapa, todo ello con el objetivo de conseguir un ratio de ejecución de instrucciones por ciclo mayor que 1.

Algunos ejemplos de procesadores superescalares son el MIPS R18000, el Alpha 21264, el HP PA-8700, el Intel Pentium 4, o el AMD Athlon.

Procesadores VLIW

Una alternativa que permite simplificar la circuitería que los procesadores superescalares dedican a la detección y gestión dinámica del paralelismo a nivel de instrucción la constituyen los procesadores VLIW (*Very Long Instruction Word*). El diseño de la circuitería de control en estos procesadores es más sencillo, ya que las operaciones que el procesador debe realizar en paralelo están codificadas en la propia instrucción. El compilador es el responsable de agrupar las operaciones que pueden emitirse a ejecución simultáneamente y crear una instrucción que contenga dichas operaciones. Así, puesto que esta detección se realiza de forma estática y previa a la ejecución, la unidad de control no necesita evaluar en tiempo de ejecución las dependencias de datos. El resultado es una circuitería de control más sencilla, que libera recursos (en forma de transistores) que en los procesadores VLIW pueden dedicarse a integrar en el chip unas memorias caché más grandes (estructuradas en un mayor número de niveles) o una cantidad más elevada de unidades funcionales. (El microprocesador Itanium 2 de Intel, el ejemplo más representativo de procesadores VLIW, integró el tercer nivel de caché dentro del chip y 4 unidades funcionales en coma flotante mucho antes que los diseños superescalares de la familia Pentium de Intel). A cambio, el grado de paralelismo entre instrucciones que puede detectarse en tiempo de compilación es más reducido del que puede extraerse en tiempo de ejecución, lo que limita el rendimiento de estos procesadores.

En la actualidad, este tipo de arquitectura es frecuente en el segmento de los procesadores empotrados, mientras que el mercado de la computación general se utilizan con mayor frecuencia los procesadores superescalares. Algunos ejemplos de procesadores VLIW son el Star Core, de Motorola y Lucent, la familia C6xxx de Texas Instruments, el Trimedia de Philips y el Itanium 2 de Intel.

Procesadores multihebra

Las arquitecturas multihebra (*multithreaded processors*) aparecieron debido a las limitaciones impuestas por las dependencias de datos y la alta latencia de la memoria en los procesadores convencionales. Se caracterizan por mantener varios flujos de ejecución en paralelo dentro del procesador. Dichos flujos pueden ser de una misma aplicación, con lo que puede reducirse su tiempo de ejecución, o de aplicaciones diferentes, con lo que puede mejorar la productividad

(*throughput*) del sistema. En concreto, las arquitecturas SMT (*Simultaneous Multithreading*) permiten que hayan instrucciones de diferentes flujos de ejecución conviviendo a la vez dentro del procesador. En cada ciclo se realiza la captación de instrucciones para las diferentes hebras, manteniendo recursos separados para cada hebra como el banco de registros o el registro de reordenación de instrucciones (*reorder buffer*). Dependiendo de la implementación, las instrucciones de las diferentes hebras comparten la cola de instrucciones y las unidades funcionales.

El ejemplo más destacable de arquitectura SMT es el Pentium 4 Xeon de Intel y las sucesivas mejoras de este procesador que también incluyen tecnología *HyperThreading*.

Procesadores multinúcleo de propósito general

Una manera de aprovechar el aumento de la escala de integración sin aumentar la complejidad arquitectónica del procesador es la integración en un único chip de varios procesadores (*núcleos*). El beneficio de este tipo de procesadores puede ser importante en aplicaciones paralelas, donde el tiempo de comunicación puede verse sensiblemente reducido al producirse ésta dentro del propio chip.

Una de las primeras propuestas de este tipo de arquitecturas fue el Standford Hydra, en el que se proponía un sistema formado por cuatro procesadores en un mismo chip compartiendo la caché de segundo nivel. Un ejemplo de procesador comercial es el Power6 de IBM, que incluye dos procesadores en un mismo chip, comunicados por una caché de segundo nivel que también reside dentro del chip. Otras compañías también han seguido este camino en los últimos años y, así, Sun ofrece su procesador Niagara 2, con 8 núcleos, Intel también integra hasta 8 núcleos en un solo chip en su arquitectura Xeon *Nehalem*, y el AMD Opteron *Istanbul* aporta hasta 6 núcleos.

Aceleradores hardware

Frente a la aproximación de propósito general, en los últimos años estamos asistiendo a una nueva época dorada de los aceleradores hardware, que aspiran a convertirse en coprocesadores en algunos casos o incluso a reemplazar al propio procesador de propósito general en otros. Dentro de esta eclosión de nuevas arquitecturas, podemos destacar el procesador Cell B.E. de IBM, con una estructura heterogénea formada por un núcleo de propósito general (PPE) conectado mediante un anillo bidireccional a 8 núcleos de cálculo (SPEs) con unidades funcionales SIMD, que proporcionan una alta velocidad de procesamiento aritmético.

Por otro lado, compañías especializadas en el diseño de hardware gráfico, como NVIDIA y ATI (ésta última absorbida por AMD), también se han mostrado receptivas a la oportunidad que se les brindaba en el mercado de la computación general, y han realizado sus propias propuestas de arquitectura (CUDA y Brooks, respectivamente) basada en la replicación masiva de unidades funcionales.

Mientras que IBM, NVIDIA y AMD/ATI sustentan sus desarrollos de nuevos diseños en un mercado más amplio, que incluye el de los videojuegos, otras compañías dirigen sus diseños específicamente al mercado de la computación de altas prestaciones. En esta línea podíamos encontrar hasta hace poco a ClearSpeed, Tilera y SciCortex.

1.3.2. Estándares de programación paralela

Los años 90 fueron testigos de una generalización del uso de arquitecturas paralelas en los entornos científicos, debido a unos costes económicos reducidos y al buen rendimiento demostrado en proyectos como el *Beowulf cluster* [151]. De la mano de esta eclosión, aparecieron cientos de propuestas destinadas a resolver el problema de la programación paralela de estas arquitecturas. Tras una década de soluciones *software*, que algunos no dudan en calificar como la década perdida de la computación paralela [115], el resultado fue la adopción de dos estándares: OpenMP [44, 45]

para el modelo de programación de variables compartidas y MPI [83, 82, 124] para el modelo de programación basado en paso de mensajes. Estas dos soluciones constituyen las herramientas más utilizadas hoy en día para la paralelización de códigos sobre arquitecturas multiprocesador con espacio de direccionamiento común y distribuido, respectivamente. Frente a otras alternativas de más bajo nivel, como la programación con hebras o la interfaz *sockets* sobre TCP/IP, OpenMP y MPI ofrecen una mayor facilidad de uso, que las hacen indudablemente más atractivas.

Aunque el auge de los procesadores multinúcleo y los sistemas heterogéneos amenaza con una nueva "Torre de Babel" de los lenguajes de programación paralela,con decenas de nuevas propuestas como Chapel, X10, UPC, Fortress, CUDA e incluso estándares como OpenCL, es todavía pronto para saber si alguna de estas herramientas logrará superar la barrera que impone la existencia de una enorme cantidad de código ya paralelizado con OpenMP y/o MPI (*legacy codes*). A continuación se ofrece una breve reseña de únicamente estas dos herramientas.

OpenMP

OpenMP [44, 45] es una herramienta de paralelización de códigos secuenciales sobre arquitecturas multiprocesador con memoria compartida, compuesta por una interfaz de programación en forma de directivas, un compilador capaz de traducir el código secuencial anotado con directivas en un programa paralelo multihebra, y un entorno de apoyo a la ejecución paralela.

La ventaja principal de OpenMP es su sencillez y su versatilidad. Por un lado, desde el punto de vista del programador, OpenMP está compuesto por un reducido número de directivas (`#pragmas`) que, utilizadas de forma adecuada, permiten transformar un código secuencial (escrito, por ejemplo, en C o en Fortran) en otro paralelo de forma gradual. Las directivas se complementan con cláusulas que facilitan al programador ajustar la ejecución paralela del código. Por otro lado, OpenMP está diseñado principalmente para explotar el paralelismo de bucles, presente en la mayor parte de códigos secuenciales, y, a partir de la versión 3.0, también en menor medida el paralelismo de tareas.

Hoy en día el estándar está muy aceptado entre la comunidad científica y entre los propios desarrolladores de software libre y comercial. Así, por ejemplo, las versiones más recientes de los compiladores de GNU e Intel, dan soporte al estándar OpenMP.

MPI

La biblioteca de comunicaciones MPI (*Message-Passing Interface*) [83, 82, 124] permite la programación de sistemas paralelos y distribuidos. Aunque está orientada hacia sistemas distribuidos y multicomputadores, también puede emplearse en multiprocesadores con memoria compartida y procesadores multinúcleo, lo cual amplía en gran medida su área de utilización. Además, es un estándar apoyado por multitud de fabricantes de ordenadores, creadores de *software* y científicos del campo de la programación distribuida y paralela. Por su alto grado de estandarización y portabilidad, esta biblioteca puede ejecutarse en un abanico muy variado de ordenadores y sistemas distribuidos y paralelos.

Una ventaja adicional de MPI es la posibilidad de desarrollar cualquier programa distribuido y paralelo con tan sólo 6 funciones. Pero si se quiere alcanzar un nivel máximo de prestaciones, es necesario conocer con profundidad la mayor parte de las funciones de que consta.

Existen diversas implementaciones de acceso público del estándar MPI, entre las que destacan MPICH del Laboratorio Nacional Argonne y la Universidad Estatal de Mississippi (EE.UU.), LAM de la Universidad Estatal de Ohio (EE.UU.) CHIMP de la Universidad de Edimburgo (Reino Unido) y OpenMPI.

1.3.3. Paquetes para la resolución iterativa de sistemas dispersos

En este apartado revisamos algunos paquetes de código para la resolución iterativa de sistemas de ecuaciones lineales dispersos. Aunque existe gran cantidad de software para este propósi-

to, presentamos una colección formada por aquellos paquetes más difundidos. Esta colección incluye paquetes consolidados que implementan los resolutores y técnicas de precondicionado más populares, junto con otros paquetes de investigación con técnicas de precondicionado que han sido desarrolladas recientemente. Excepto cuando se apunte lo contrario, todos estos paquetes incluyen soporte para procesamiento paralelo, utilizan el modelo de programación basado en paso de mensajes, y están dirigidos a computadores masivamente paralelos, es decir, arquitecturas multiprocesador con espacio de direccionamiento distribuido.

PETSc

El paquete PETSc [19, 20, 21] ha sido desarrollado en el laboratorio nacional de Argonne (EE.UU) con el principal propósito de dotar al usuario de las herramientas necesarias para construir software paralelo escalable para la resolución de aplicaciones de ciencia e ingeniería, con especial atención a aquellas que están gobernadas por EDPs. PETSc incluye implementaciones eficientes de los métodos basados en la generación de subespacios de Krylov y de los precondicionadores más populares, como la factorización ILU con nivel de llenado o basada en umbral de magnitud, así como interfaces para el acceso a una gran variedad de software externo, como es el caso de los paquetes Trilinos/ML e Hypre (presentados más abajo). También incluye precondicionadores de descomposición de dominios, como el precondicionador AS de un sólo nivel, y precondicionadores multinivel para problemas específicos como la discretización de EDPs en mallas regulares. Para problemas menos específicos, PETSc incluye una infraestructura que permite al usuario configurar los bloques básicos (por ejemplo, los operadores de prolongación y restricción) de sus propios precondicionadores AMG. El paquete está escrito en el lenguaje C, con un estilo de programación orientado a objetos, y utiliza MPI como biblioteca de paso de mensajes.

Trilinos

El paquete Trilinos [92] ha sido desarrollado en el laboratorio nacional Sandia (EE.UU) con el objeto de integrar resolutores y bibliotecas paralelas en un marco de trabajo orientado a objetos. Trilinos está compuesto por varias bibliotecas desarrolladas de forma independiente, que pueden ser utilizadas de forma aislada o en conjunción con otros paquetes que proporcionan interfaces para su integración. El paquete Ifpack proporciona interfaces orientadas a objetos para precondicionadores estilo Jacobi y basados en la factorización ILU; el paquete ML [73] para precondicionadores paralelos AMG basados en *smoothed aggregation* [160, 161]; y los paquetes AztecOO, que proporciona una interfaz orientada a objetos a la popular biblioteca Aztec [157], y más recientemente Belos, contienen implementaciones de los métodos basados en la generación de subespacios de Krylov. Los núcleos computacionales han sido escritos en el lenguaje C, las interfaces de acceso a estos núcleos en C++, y el paquete utiliza MPI como biblioteca de paso de mensajes.

Hypre

Hypre es una biblioteca desarrollada en el laboratorio nacional Lawrence Livermore (EE.UU), diseñada principalmente para la resolución de sistemas de ecuaciones dispersos y de gran dimensión en sistemas masivamente paralelos. La biblioteca implementa precondicionadores paralelos basados en la factorización ILU (Euclid [96]), precondicionadores paralelos basados en la factorización aproximada de la inversa (ParaSails [49, 51]) y precondicionadores paralelos AMG (Boomer AMG [91]). La biblioteca está escrita en C, aunque proporciona interfaces para C++, Fortran y Python, y utiliza MPI para el envío de mensajes entre los procesos.

pARMS

La biblioteca pARMS [126] ha sido desarrollada en la Universidad de Minnesota (EE.UU). A partir del enfoque de los métodos descomposición de dominios, la biblioteca implementa precondicionadores paralelos basados en la factorización ILU multinivel y resolutores basados en la generación de subespacios de Krylov para sistemas lineales dispersos distribuidos [138]. Aunque los precondicionadores basados en la factorización ILU multinivel no son precondicionadores AMG en un sentido "estricto" [37, 38, 39, 134], guardan ciertas conexiones con estos métodos [111, 140]. La biblioteca está escrita en el lenguaje C y Fortran77, y utiliza MPI para las comunicaciones entre procesos.

ILUPACK

ILUPACK [35] es una biblioteca de precondicionado que proporciona métodos para la resolución iterativa de sistemas de ecuaciones lineales dispersos y de gran dimensión. Las técnicas de precondicionado de la biblioteca se basan en una variante multinivel de la factorización ILU que tiene por objeto controlar el crecimiento de la magnitud de las entradas de la inversa de los factores [34]. Esta factorización se aplica como precondicionador del sistema lineal, y el sistema precondicionado se resuelve empleando alguno de los métodos basados en la generación de subespacios de Krylov. La biblioteca proporciona técnicas de precondicionado y métodos iterativos específicos para cada tipo de sistema, considerando los casos SDP, simétrico indefinido, y no simétrico, e incluye rutinas para matrices reales y complejas. Además de las rutinas estándar de reordenación para reducción de llenado (como por ejemplo, MLND [102], RCM [56] o AMD [14]), la biblioteca también incluye las estrategias de ordenación de los métodos multinivel ARMS [140] (como por ejemplo, INDSET o ddPQ), y rutinas de ordenación para situar los elementos de mayor magnitud de la matriz en su diagonal principal [31, 63, 64, 88, 123, 141]. Estos últimos métodos son prácticamente obligatorios para mejorar la estabilidad numérica de la factorización incompleta de matrices simétricas indefinidas y generales. La biblioteca ILUPACK está escrita en C y Fortran77, y en la actualidad *no incluye precondicionadores ni resolutores paralelos*.

1.4. Motivación e interés

La resolución *eficiente* de sistemas de ecuaciones lineales dispersos y de gran dimensión es uno de los mayores desafíos de la computación científica. Las simulaciones actuales requieren el uso de modelos computacionales tridimensionales cada vez más elaborados, lo que se traduce en un aumento del tamaño de los sistemas y del tiempo de simulación. La resolución de estos sistemas es típicamente un cuello de botella de todo el proceso de simulación, motivando el desarrollo de métodos de resolución eficientes y escalables; es decir, métodos cuyas demandas computacionales y de almacenamiento sólo crezcan moderadamente con el tamaño del sistema. Las nuevas arquitecturas de computación de altas prestaciones posibilitan la resolución de aplicaciones cada vez más complejas, y abren nuevos desafíos, como el desarrollo de algoritmos *paralelos* de resolución capaces de aprovechar eficientemente el paralelismo explícito en forma de múltiples núcleos de procesamiento. De hecho, la demanda de algoritmos y arquitecturas paralelas es cada vez más frecuente en aplicaciones tridimensionales, dado que los sistemas de ecuaciones lineales que se necesitan resolver están compuestos típicamente por cientos de millones de ecuaciones e incógnitas.

Para la resolución eficiente de sistemas de ecuaciones dispersos y de gran dimensión consideraremos el uso de métodos iterativos basados en la generación de subespacios de Krylov, combinados con las técnicas de precondicionado de la biblioteca ILUPACK. Los precondicionadores de esta biblioteca se basan en una variante multinivel de la factorización ILU que es especialmente robusta y eficiente para la resolución de sistemas de ecuaciones que surgen de la

discretización de EDPs (o, al menos, con propiedades algebraicas relacionadas), aunque también puede ser aplicada a sistemas de ecuaciones lineales arbitrarios. Recientemente [31] se ha demostrado que el enfoque de las técnicas de precondicionado de ILUPACK imita a los métodos AMG [37, 38, 39, 134], que explotan la posibilidad que se da en los problemas gobernados por EDPs para construir un sistema más "grueso" (*coarser*), y mantener en éste los valores propios más pequeños (típicamente referidos como "bajas frecuencias") del sistema original. Sin embargo, el enfoque algebraico más general de los métodos de ILUPACK permite aplicar esta clase de factorizaciones ILU multinivel a sistemas más generales que los métodos AMG. En particular, ILUPACK ha sido aplicado con éxito para resolver problemas tridimensionales de interés científico altamente indefinidos, como el modelo de localización de Anderson [141], o la EDP de Helmholtz en medios heterogéneos [31]. En todas estas aplicaciones, una versión paralela de la biblioteca ILUPACK permitiría, por una parte, reducir de forma significativa el tiempo de simulación, y, por otra, realizar simulaciones bajo condiciones que requieren mayor coste computacional y de almacenamiento. Por ejemplo, en el caso de la EDP de Helmholtz, interesa realizar simulaciones con longitudes de onda cada vez más pequeñas, lo que en cualquier método estándar de discretización se traduce en una reducción del tamaño de la malla y, por tanto, en un aumento del tamaño del sistema [31].

En la actualidad, la biblioteca ILUPACK *no incluye precondicionadores ni resolutores paralelos* y, por tanto, en esta tesis proponemos el desarrollo, análisis e implementación de algoritmos iterativos paralelos de resolución basados en las técnicas de precondicionado de la biblioteca ILUPACK. En particular, como primer paso para cubrir esta carencia, consideramos el desarrollo y análisis de algoritmos paralelos para multiprocesadores con memoria compartida y, para su implementación, el estándar OpenMP, principalmente por las siguientes razones. En primer lugar, estos multiprocesadores están típicamente compuestos por 16-64 microprocesadores, proporcionando la capacidad computacional suficiente para muchas aplicaciones de interés científico. Por ejemplo, en el caso del modelo de localización de Anderson, la simulación requirió, para el problema de mayor dimensión considerado en [141], tres días utilizando un sólo procesador Itanium 2 en una plataforma paralela con 108 GB de memoria. Un multiprocesador con memoria compartida combinado con un algoritmo paralelo de resolución escalable hasta 16 procesadores permitiría resolver el sistema en tan sólo unas horas. En segundo lugar, los multiprocesadores con memoria compartida constituyen típicamente los bloques básicos de multiprocesadores más potentes, como los de memoria distribuida, habilitando el aprovechamiento de un primer tipo/nivel de paralelismo en este último tipo de multiprocesadores. En tercer lugar, OpenMP proporciona una interfaz para el desarrollo de aplicaciones natural, simple y flexible para implementar códigos paralelos sobre multiprocesadores con memoria compartida y, además, los multiprocesadores con memoria compartida se programan usando el mismo modelo de programación que los más modernos microprocesadores multinúcleo de propósito general. Por lo tanto, el uso de herramientas estándar como OpenMP permite ejecutar los códigos sobre este tipo de microprocesadores, que actualmente dominan el mercado de la computación de altas prestaciones.

1.5. Objetivos

El trabajo realizado en esta tesis tiene como objetivo general *analizar el paralelismo de tareas intrínseco en los métodos de la biblioteca ILUPACK y evaluar experimentalmente, a partir de implementaciones paralelas, si este grado de paralelismo es suficiente para su ejecución eficiente sobre multiprocesadores con memoria compartida.*

Para alcanzar el objetivo general, el trabajo realizado persigue la consecución de los siguientes objetivos específicos:

- Revisar el estado del arte para comprender la estructura y semántica de las técnicas de precondicionado basadas en la factorización ILU multinivel de ILUPACK, su conexión con

otros métodos de factorización ILU y/o multinivel, y los fundamentos y propiedades de los métodos de resolución basados en la generación de subespacios de Krylov.

- Identificar paralelismo de tareas en las técnicas de precondicionado basadas en la factorización ILU multinivel y en la resolución iterativa del sistema precondicionado. Para ello, es necesario investigar la concurrencia inherente a las computaciones que realizan estos métodos, para posteriormente representarla mediante una abstracción que identifique tareas concurrentes y sus dependencias.

- Diseñar estrategias de planificación y mapeado de tareas a procesos (procesadores) conscientes de las propiedades particulares de las computaciones que realizan los métodos (por ejemplo, grado de irregularidad o de intensidad aritmética) y la organización de la arquitectura de destino.

- Implementar los algoritmos paralelos siguiendo los principios de diseño de la biblioteca ILUPACK (estructuras de datos, gestión de memoria, prototipos y organización de las rutinas), utilizando herramientas estándar para la programación de arquitecturas multiprocesador con espacio de direccionamiento común.

- Comparar la eficiencia y robustez del enfoque escogido para la paralelización con el método precondicionado secuencial, considerando para ello las siguientes métricas complementarias: tiempo de resolución, velocidad de convergencia, calidad de la solución y consumo de memoria. Este objetivo también pretende, por una parte, evaluar hasta qué punto el enfoque de paralelización preserva las propiedades numéricas (semántica) del método secuencial y, por otra, estudiar el compromiso entre grado de paralelismo y sobrecoste computacional/de almacenamiento al aumentar el número de tareas en las que se puede dividir el problema.

- Seleccionar aplicaciones reales y/o académicas, principalmente gobernadas por EDPs, para llevar a cabo el estudio perseguido por el objetivo previo en un multiprocesador con memoria compartida comercial.

1.6. Estructura de la memoria

Además de este primer capítulo introductorio, la memoria de tesis consta de cuatro capítulos adicionales. En el segundo, tercer y cuarto capítulo presentamos las aportaciones principales de la tesis. En particular, en el segundo capítulo detallamos la paralelización para multiprocesadores con memoria compartida de las técnicas de precondicionado de la biblioteca ILUPACK, mientras que en el tercer capítulo discutimos la paralelización del método de los gradientes conjugados precondicionados, y en el cuarto capítulo presentamos los experimentos y analizamos los resultados obtenidos al emplear nuestros algoritmos paralelos para resolver varios problemas de aplicación en el multiprocesador SGI Altix 350 (consultar Apéndice B). Para concluir, en el último capítulo resumimos las conclusiones más relevantes de la investigación junto con una serie de líneas abiertas de trabajo. A continuación describimos de forma más detallada el contenido de cada capítulo.

En lo que resta del presente capítulo, presentamos la notación y algunas definiciones básicas necesarias para los desarrollos del resto de capítulos (sección 1.7)

En el Capítulo 2 incluimos, en primer lugar, una revisión del estado del arte en materia de precondicionadores basados en la factorización ILU. Esta revisión presenta paulatinamente los métodos básicos de factorización ILU, sus fundamentos, objetivos, ventajas potenciales y limitaciones, y algunas notas referentes a su implementación y paralelización. La revisión también discute la problemática de la existencia y estabilidad de las factorizaciones ILU, el efecto que

tienen las estrategias de reordenación sobre la calidad de éstas cuando actúan como precondicionadores, y presenta los métodos multinivel, y su conexión con las factorizaciones ILU multinivel. Posteriormente, se detallan las técnicas de precondicionado de la biblioteca ILUPACK, y cómo sus bloques básicos configuran un método de precondicionado multinivel más robusto, eficiente y escalable que los métodos clásicos de factorización ILU.

A continuación, en el Capítulo 2 introducimos nuestro enfoque para la paralelización de las técnicas de precondicionado de ILUPACK. Esta aproximación se basa en una fase inicial que particiona el grafo de adyacencia de la matriz del sistema para identificar un conjunto de tareas concurrentes y sus dependencias. El enfoque impone un cierto orden en el que deben eliminar las incógnitas del sistema reordenado, transformando el algoritmo básico de ILUPACK con el propósito de identificar un alto grado de concurrencia en el proceso de construcción del precondicionador. En el capítulo se detalla el algoritmo transformado, se analiza la concurrencia inherente a las computaciones que realiza, y se indica cómo aprovechar esta concurrencia de forma eficiente. Además, también se justifican las estrategias de planificación y mapeado de tareas a procesos, y se analiza el rendimiento del algoritmo paralelo cuando se aplica para resolver una aplicación irregular de ejemplo, la EDP de difusión en estado estacionario bidimensional con coeficientes discontinuos.

En el Capítulo 3 se revisa, en primer lugar, el método de los gradientes conjugados, sus propiedades, fundamentos, estructura, convergencia y precondicionado. La estructura lógica de los precondicionadores paralelos construidos por nuestro algoritmo permite aprovechar paralelismo de forma natural en su aplicación. En el capítulo se analiza también la concurrencia que se puede explotar en la aplicación de los precondicionadores paralelos basados en la factorización ILU multinivel, y se indica cómo aprovechar esta concurrencia de forma eficiente. Posteriormente, presentamos las estrategias de mapeado y planificación de las tareas en las que se descompone la aplicación del precondicionador, y la paralelización del resto de operaciones del método CG, es decir, productos escalares, actualizaciones de vectores y producto de una matriz dispersa por un vector. En nuestro enfoque, la paralelización de estas operaciones se realiza coordinada con la aplicación del precondicionador, con el objeto de aumentar la localidad de referencia y reducir el sobrecoste asociado al tráfico de datos en el sistema de memoria del multiprocesador. Por último, en el capítulo se estudia el rendimiento del método paralelo de los gradientes conjugados precondicionados cuando se aplica para resolver los sistemas de ecuaciones que surgen en la aplicación de ejemplo empleada en el Capítulo 2.

En el Capítulo 4 se analizan los resultados computacionales obtenidos al aplicar nuestros algoritmos paralelos para resolver los sistemas de ecuaciones lineales dispersos y de gran dimensión que aparecen ligados a varias EDPs tridimensionales en el multiprocesador SGI Altix 350. Asimismo, para considerar problemas más irregulares, que surjan en diversas áreas de aplicación, también aplicamos nuestro enfoque para resolver en paralelo una miscelánea de sistemas de ecuaciones lineales de gran dimensión obtenidos de la colección de matrices dispersas de la Universidad de Florida [55]. Los experimentos realizados ponen de manifiesto la eficiencia y robustez del enfoque escogido para la paralelización en comparación con el método precondicionado secuencial, empleando para ello diversas métricas como el tiempo de resolución, la aceleración paralela, la velocidad de convergencia (número de iteraciones), la calidad (precisión) de la solución obtenida, la memoria consumida por el método para construir el precondicionador, y el tamaño ocupado por el precondicionador en memoria (número de elementos no nulos), con distintos tamaños de matrices y número de procesadores. En este capítulo también se proporcionan evidencias detalladas que justifican en la medida de lo posible los resultados obtenidos, comparando, por ejemplo, la jerarquía multinivel construida por el algoritmo de ILUPACK y nuestro algoritmo paralelo. Asimismo, también se estudia el efecto que tienen los algoritmos paralelos de particionado de matrices dispersas incluidos en paquetes de software actuales sobre el desempeño de los algoritmos evaluados.

En el Capítulo 5 se enumeran las conclusiones y aportaciones más relevantes del trabajo

realizado junto con una serie de líneas abiertas de investigación.

Por último, en el Apéndice A describimos los formatos de almacenamiento de matrices dispersas más comunes, y en el Apéndice B describimos detalladamente la arquitectura del multiprocesador con memoria compartida en la que se han realizado los experimentos de la tesis.

1.7. Notación y definiciones

En esta sección se presenta la notación usada en esta memoria junto con una miscelánea de definiciones básicas necesarias para los desarrollos de los capítulos siguientes.

Escalares y vectores

- Las letras griegas minúsculas, como α, β o γ, denotan escalares reales.
- Un vector real de n componentes se denota con una letra minúscula, $x \in \mathbb{R}^n$, donde $\mathbb{R}^n$ es el conjunto de vectores reales de tamaño n. Como excepción, las letras minúsculas desde la i hasta la n no se utilizan para referirse a vectores; en su lugar, denotan escalares enteros.
- La componente i-ésima de un vector x la denotamos con la letra x y el subíndice i, es decir, x_i.
- Los vectores fila se denotan con una letra minúscula y el superíndice T, por ejemplo, b^T, x^T, y^T, z^T, ...
- La expresión 0_k se refiere al vector nulo de tamaño k.

Matrices

- Una matriz real de m filas y n columnas se denota con una letra mayúscula, $A \in \mathbb{R}^{m,n}$, donde $\mathbb{R}^{m,n}$ es el conjunto de las matrices reales de tamaño $m \times n$.
- La entrada de la matriz A situada en la fila i-ésima y columna j-ésima se denota con la letra minúscula correspondiente y el subíndice ij, es decir, a_{ij}, o con la letra mayúscula entre paréntesis y el subíndice ij, es decir, $(A)_{ij}$.
- La columna k-ésima de A se denota con la letra minúscula correspondiente y el subíndice k, es decir, a_k, y la fila k-ésima con la letra minúscula correspondiente, el subíndice k y el superíndice T, es decir, a_k^T. Nótese que, dependiendo del contexto, a_k también puede referirse a la componente k-ésima del vector a.
- La letra L se utilizará para denotar una matriz triangular inferior y la letra U para una matriz triangular superior; una matriz A es triangular inferior si $a_{ij} = 0$, para $i < j$, y triangular superior si $a_{ij} = 0$, para $i > j$.
- La letra $I \in \mathbb{R}^{n,n}$ se refiere a la matriz *identidad* de orden n, y e_k y e_k^T a la columna y fila k-ésima de I, respectivamente. La matriz identidad es una matriz con $(I)_{ii} = 1$, e $(I)_{ij} = 0$ para $i \neq j$.
- La transpuesta A^T de una matriz real A se define como $(A^T)_{ij} = a_{ji}$.

Operaciones matriciales y vectoriales básicas

- La suma de matrices, $C = A + B$, se define como $c_{ij} = a_{ij} + b_{ij}$, y la multiplicación de un escalar α por una matriz A, $C = \alpha A$, como $c_{ij} = \alpha a_{ij}$; la suma de matrices requiere que A y B tengan el mismo número de filas y columnas.

- La multiplicación de matrices, $C = AB$, se define como

$$c_{ij} = \sum_{k=1}^{s} a_{ik}\ b_{kj}, \tag{1.5}$$

  donde $C \in \mathbb{R}^{m,n}$, $A \in \mathbb{R}^{m,s}$, y $B \in \mathbb{R}^{s,n}$; la multiplicación de matrices requiere que el número de columnas de A sea igual al número de filas de B.

- El producto escalar o interno de dos vectores $x \in \mathbb{R}^n$ e $y \in \mathbb{R}^n$ es el escalar $x^T y = \sum_{i=1}^{n} x_i y_i$ y, el producto externo, la matriz xy^T cuya entrada $(xy^T)_{ij}$ se define como $(xy^T)_{ij} = x_i y_j$; las operaciones de la forma $A \leftarrow A + xy^T$ se denominan actualizaciones de rango 1.

- Si X e Y son matrices con k columnas, entonces XY^T se denomina producto externo de rango k, y las operaciones de la forma $A \leftarrow A + XY^T$, actualizaciones de rango k.

Matrices por bloques

Una matriz por bloques es una matriz cuyas entradas pueden ser matrices o vectores, y se define a partir de un *particionado* de las filas y/o columnas de una matriz. Por ejemplo, si las filas de una matriz $A \in \mathbb{R}^{m,n}$ se particionan en r subconjuntos de tamaños $m_1, m_2, \ldots, m_r$, y sus columnas en c subconjuntos de tamaños $n_1, n_2, \ldots, n_c$, entonces A se puede escribir como una matriz por bloques de tamaño $r \times c$:

$$A = \begin{pmatrix} A_{11} & \ldots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{r1} & \ldots & A_{rc} \end{pmatrix}, \tag{1.6}$$

donde A_{ij} es un bloque de A de tamaño $m_i \times n_j$. Dos matrices por bloques A y B se pueden sumar por bloques, $C = A+B$, si tienen la misma dimensión y sus filas y columnas se particionan de forma idéntica; las filas y columnas de C se particionan conforme a las filas y columnas de A y B. Dos matrices por bloques se pueden multiplicar por bloques, $C = AB$, si el número de columnas de A coincide con el número de filas de B; las columnas de A se particionan conforme a las filas de B; y las filas y columnas de C se particionan conforme a las filas de A y columnas de B, respectivamente. Si c denota el número de subconjuntos en el que se particionan las columnas de A y filas de B, entonces el producto por bloques de A y B puede calcularse como

$$C_{ij} = \sum_{k=1}^{c} A_{ik}\ B_{kj}. \tag{1.7}$$

Algunas definiciones

- Dos vectores x e y son ortogonales si su producto escalar es nulo, es decir, si $x^T y = 0$.

- Un conjunto de vectores $v_1, v_2, \ldots, v_n$ es linealmente independiente si $\sum_{i=1}^{n} \alpha_i v_i = 0$ implica que $\alpha_1 = \alpha_2 = \ldots = \alpha_n = 0$.

- El subespacio generado por los vectores $v_1, v_2, \ldots, v_n$ se denota como $\text{span}\{v_1, v_2, \ldots, v_n\}$, y se define como el conjunto de vectores que se pueden obtener como una combinación de $v_1, v_2, \ldots, v_n$; es decir, $\text{span}\{v_1, v_2, \ldots, v_n\} = \{x : x = \sum_{i=1}^{n} \alpha_i v_i\}$.

- La imagen de una matriz A es el subespacio generado por sus columnas.

- El rango de una matriz es el tamaño de aquel subconjunto formado por el mayor número de columnas de A linealmente independientes.

- Una matriz cuadrada de tamaño $n \times n$ es *singular* si su rango es inferior a n, y no singular en caso contrario. Una matriz rectangular de tamaño $m \times n$ es de rango deficiente si su rango es es menor que mín(m, n), y de rango completo en caso contrario.
- Una matriz cuadrada real es *simétrica* si $A = A^T$, *simétrica definida positiva* si $A = A^T$ y el producto $x^T Ax > 0$ para cualquier vector x que no sea el vector nulo, y *simétrica semidefinida positiva* si $A = A^T$ y $x^T Ax \geq 0$ para cualquier vector x.
- La inversa de una matriz cuadrada A se denota como A^{-1}, y es una matriz que cumple $AA^{-1} = A^{-1}A = I$. La inversa sólo existe si A es no singular.
- Una matriz es diagonal dominante por filas si $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}|$, y estrictamente diagonal dominante por filas si $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$. Una matriz es (estrictamente) diagonal dominante por columnas si A^T es (estrictamente) diagonal dominante por filas. Una matriz cuadrada estrictamente diagonal dominante es no singular.
- Una matriz cuadrada general, $A\mathbb{R}^{n,n}$, es una M-matriz si y solo si cumple las siguientes cuatro propiedades:
  1. $a_{ii} > 0$, para $k = 1, \ldots, n$.
  2. $a_{ij} \leq 0$, para $i \neq j,\ i, j = 1, \ldots, n$.
  3. A es no singular.
  4. $A^{-1} \geq 0$.
- Una matriz cuadrada general, $A\mathbb{R}^{n,n}$, es una H-matriz, si la matriz H, definida como $H = (h_{ij}), h_{ii} = |a_{ii}|, h_{ij} = -|a_{ij}|, i \neq j$, es una M-matriz.
- Una matriz de permutación P es aquella que se obtiene reordenando las filas (o columnas) de la matriz identidad. Cualquier fila o columna de P sólo tiene una entrada no nula e igual a la unidad. En esta memoria utilizaremos las letras P o Q, o letras griegas mayúsculas, como Π, para referirnos a matrices de permutación.

Normas vectoriales y matriciales básicas

- La norma-1 de un vector $x \in \mathbb{R}^n$ se define como $\|x\|_1 = \sum_{i=1}^{n} |x_i|$, su norma-2 como $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$, y su norma-infinito como $\|x\|_\infty = \text{máx}_{i=1,2,\ldots,n} |x_i|$, donde $|x_i|$ es el valor absoluto de x_i.
- Si $A \in \mathbb{R}^{n,n}$ es una matriz simétrica definida positiva, entonces la norma A del vector $x \in \mathbb{R}^n$ se define como $\|x\|_A = \sqrt{x^T Ax}$.
- La norma-1 de una matriz $A \in \mathbb{R}^{m,n}$ se define como $\|A\|_1 = \text{máx}_{i=1,2,\ldots,n} \|a_i\|_1$ y su norma-infinito como $\|A\|_\infty = \text{máx}_{i=1,2,\ldots,m} \|a_i^T\|_1$.
- $\|A\|$ denota "alguna norma matricial" cuando la clase concreta de norma no sea relevante para su contexto.

Patrones de elementos no nulos y conjuntos

Utilizaremos letras caligráficas como $\mathcal{A}$ para referirnos al patrón de elementos no nulos de una matriz *dispersa* A, es decir, al conjunto de posiciones en las que residen su entradas no nulas. Por tanto, la cardinalidad del conjunto $\mathcal{A}$ es el número de entradas no nulas de A, es decir, $nnz(A) = |\mathcal{A}|$. Las letras caligráficas también se utilizarán para denotar conjuntos, independientemente de si éstos se refieren o no al conjunto de posiciones de las entradas no nulas de una matriz dispersa.

Relación entre las matrices dispersas y los grafos

La teoría de grafos proporciona herramientas muy potentes que son de gran utilidad para la resolución de problemas matriciales dispersos. Todas estas herramientas se apoyan en la relación que existe entre los grafos y las matrices dispersas. Esta conexión se materializa a través del grafo de adyacencia $G(A)$ correspondiente a una matriz A. En general, un grafo $G = \{V, E\}$ se define como una tupla formada por dos conjuntos: el conjunto de vértices

$$V = \{v_1, v_2, \ldots, v_n\}, \tag{1.8}$$

y el conjunto de aristas

$$E \subseteq V \times V. \tag{1.9}$$

En el contexto particular de las matrices dispersas, el grafo de adyacencia $G(A) = \{V, E\}$ de una matriz A captura su patrón de elementos no nulos, de modo que los conjuntos de vértices y aristas están definidos como $V = \{1, \ldots, n\}$ y $E = \{(i, j) : a_{ij} \neq 0 \wedge i \neq j\}$, respectivamente. Este grafo es dirigido para matrices no simétricas y no dirigido para matrices simétricas. La Figura 1.1 muestra, a modo de ejemplo, el patrón de dispersidad de una matriz dispersa simétrica y el grafo de adyacencia correspondiente. Los bucles $(i, i) \in E$ correspondientes a las entradas de la diagonal principal no se suelen representar gráficamente.

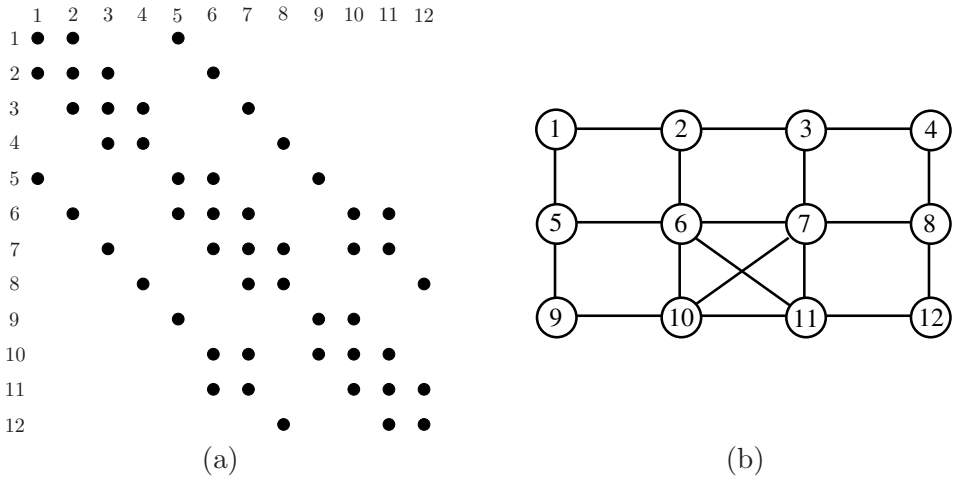


Figura 1.1: (a) Patrón de dispersidad de una matriz dispersa A. (b) Grafo de adyacencia $G(A)$ correspondiente.

En la literatura se dispone de multitud de algoritmos de tratamiento de grafos que se pueden aplicar, con diferentes propósitos, para resolver problemas matriciales. Todos estos algoritmos se suelen traducir en forma de permutaciones que transforman estratégicamente la estructura dispersa de la matriz original. Por ejemplo, existen métodos numéricos especiales para la resolución de problemas matriciales banda que superan claramente a los métodos generales. En este sentido, el algoritmo RCM [62] proporciona una ordenación del grafo asociado a una matriz que minimiza su ancho de banda, habilitando la utilización de métodos numéricos específicos para este tipo de matrices. Otra familia de algoritmos particiona el grafo de adyacencia de tal forma que el problema matricial correspondiente puede ser resuelto en un entorno de computación paralela minimizando el coste de las comunicaciones; los paquetes de particionado SCOTCH [128] o METIS [103] resultan especialmente eficientes para este propósito. Un tercer grupo es el método basado en grafos diseñado por Hagemann y Schenk [88], que mejora la estabilidad numérica de los métodos directos de resolución de sistemas de ecuaciones lineales simétricos indefinidos.

Capítulo 2

Cálculo paralelo de precondicionadores LDL^T multinivel

En este capítulo introducimos nuestro enfoque para la paralelización de las técnicas de precondicionado de ILUPACK. Esta aproximación se basa en una etapa inicial que particiona el grafo de adyacencia de la matriz del sistema para identificar un conjunto de tareas concurrentes y sus dependencias. El enfoque impone un cierto orden en el que deben eliminar las incógnitas del sistema reordenado, transformando el algoritmo básico de ILUPACK con el propósito de identificar un alto grado de concurrencia en el proceso de construcción del precondicionador.

La estructura del capítulo es la siguiente. En la Sección 2.1 incluimos una revisión del estado del arte en materia de precondicionadores basados en la factorización ILU. Esta revisión presenta paulatinamente los métodos básicos de factorización ILU, sus fundamentos, objetivos, potencialidades y limitaciones, y algunas notas referentes a su implementación. La revisión también discute la problemática de la existencia y estabilidad de las factorizaciones ILU, el efecto que tienen las estrategias de reordenación sobre la calidad de éstas cuando actúan como precondicionadores, y presenta los métodos multinivel y su conexión con las factorizaciones ILU multinivel. Posteriormente, en la Sección 2.2 se detallan las técnicas de precondicionado de la biblioteca ILUPACK, y cómo sus bloques básicos configuran un método de precondicionado multinivel más robusto, eficiente y escalable que los métodos clásicos de factorización ILU. Por último, en la Sección 2.3 se detalla cómo exponer concurrencia mediante una transformación del método secuencial de ILUPACK, se analiza la concurrencia inherente a las computaciones que realiza esta transformación, y se indica cómo aprovechar esta concurrencia de forma eficiente. Además, también se justifican las estrategias de planificación y mapeado de tareas a procesos, y se analiza el rendimiento del algoritmo paralelo cuando se aplica para resolver una aplicación irregular de ejemplo, la EDP de difusión en estado estacionario con coeficientes discontinuos.

2.1. Precondicionadores basados en la factorización ILU

Los factores triangulares obtenidos como resultado de la factorización LU de una matriz dispersa A son, con frecuencia, considerablemente más densos que A, debido al llenado que se produce en el proceso de factorización. A pesar de que se han desarrollado técnicas heurísticas de reordenación *altamente eficaces* para reducir el llenado [56, 62], los métodos directos no son viables para resolver sistemas de ecuaciones de dimensión extremadamente grande, debido a los recursos computacionales y de almacenamiento que éstos requieren. Una solución simple e intuitiva para abordar este problema consiste en descartar elementos de las matrices involucradas en el curso de la factorización, obteniendo, por tanto, dos factores triangulares, $\tilde{L} \in \mathbb{R}^{n,n}$ triangular inferior unidad y $\tilde{U} \in \mathbb{R}^{n,n}$ triangular superior, tal que $A \approx \tilde{L}\tilde{U}$. El proceso que obtiene esta aproximación se denomina factorización ILU (*Incomplete LU*) de A, y $\tilde{L}$, $\tilde{U}$ se conocen como factores incompletos o aproximados de A. Posteriormente, $M = \tilde{L}\tilde{U}$ se puede emplear como pre-

condicionador de (1.1), con la esperanza de que $M^{-1}A \approx I$. Empleando esta técnica se pueden derivar precondicionadores simples, pero con frecuencia eficaces para resolver iterativamente el sistema de ecuaciones lineales (1.1).

En el Apartado 2.1.1 introducimos incrementalmente los métodos y técnicas *básicas* que se pueden emplear para derivar precondicionadores a partir de factorizaciones ILU de A. En el Apartado 2.1.2 discutimos aspectos relacionados con la existencia y estabilidad de las factorizaciones ILU, como las condiciones *suficientes* bajo las cuales A admite una factorización ILU, los factores que pueden condicionar su existencia y estabilidad, y las técnicas a emplear para atenuarlos y/o eliminarlos. Aunque los métodos básicos de factorización ILU pueden ser poco eficaces cuando se emplean *aisladamente*, su fiabilidad, robustez y eficacia puede mejorar significativamente cuando se combinan con técnicas como las descritas en el Apartado 2.1.3, y/o cuando se integran como bloques básicos en métodos más sofisticados, como por ejemplo los métodos multinivel, presentados en el Apartado 2.1.4.

2.1.1. Fundamentos de los métodos y técnicas básicas de precondicionado

Eliminación Gaussiana y su variante aproximada

La eliminación Gaussiana [76, 164] es un proceso que transforma la matriz de coeficientes A en una matriz con estructura triangular superior. Comenzando con $A_0 = A$, el proceso aplica, en cada iteración $k = 1, 2, \ldots, n-1$, una *transformación Gaussiana* a $A_{k-1} \in \mathbb{R}^{n,n}$, que es la matriz resultante de aplicar a A_0 las $k-1$ transformaciones anteriores. Esta transformación *elimina* (anula) los elementos de la columna k-ésima de A_{k-1} que están por debajo de la diagonal principal. Denotando los elementos de la matriz resultante de aplicar k transformaciones Gaussianas como $A_k = (a_{ij}^{(k)})$, el efecto de la transformación efectuada en la iteración k-ésima se puede expresar en forma del siguiente producto matricial

$$A_k = L_k^{-1} A_{k-1}, \tag{2.1}$$

con

$$L_k^{-1} = I - \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0_k \\ a_{*k}^{(k-1)} \end{pmatrix} e_k^T, \tag{2.2}$$

y

$$a_{*k}^{(k-1)} = \begin{pmatrix} a_{k+1k}^{(k-1)} & \ldots & a_{nk}^{(k-1)} \end{pmatrix}^T \tag{2.3}$$

el vector en $\mathbb{R}^{n-k,1}$ formado por los elementos situados por debajo de la diagonal principal de la columna k-ésima de A_{k-1}. Es fácil demostrar que con L_k^{-1} definida en (2.2), con $k = 1, 2, \ldots, n-1$, entonces la aplicación de $n-1$ transformaciones Gaussianas a $A_0 = A$ reduce esta matriz a forma triangular superior:

$$L_{n-1}^{-1} \cdots L_1^{-1} A = U. \tag{2.4}$$

Además,

$$L_k = I + \frac{1}{a_{kk}^{(k-1)}} \begin{pmatrix} 0_k \\ a_{*k}^{(k-1)} \end{pmatrix} e_k^T \tag{2.5}$$

y, como consecuencia, la matriz

$$L = L_1 \cdots L_{n-1} \tag{2.6}$$

es triangular inferior unidad, y $A = LU$.

Empleando las expresiones (2.1)-(2.6), la derivación de la variante algorítmica orientada a la derecha (*right-looking*) de la factorización LU [84], mostrada en el Algoritmo 2.1, es inmediata. Los elementos de L y de U se denotan, respectivamente, como l_{ij} y u_{ij}. En cada iteración

del bucle externo se obtiene la parte triangular inferior estricta de la columna k-ésima de L (línea 3) conforme a (2.5), y la fila k-ésima de U (línea 4), que coincide con la fila k-ésima de A_{k-1}. Posteriormente, los dos bucles anidados internos se encargan de aplicar la transformación Gaussiana correspondiente a la iteración k-ésima, es decir, de calcular A_k conforme a (2.1).

En este punto del desarrollo es posible definir el concepto de *llenado* en el proceso de factorización LU de una matriz *dispersa* A. Consideremos que $\mathcal{A}$, $\mathcal{A}_k$, $\mathcal{L}$ y $\mathcal{U}$ son, respectivamente, los patrones de elementos no nulos de A, A_k y los correspondientes factores L y U calculados mediante el Algoritmo 2.1. La actualización de la línea 7 del Algoritmo 2.1 produce un *nuevo* elemento (i, j) en el conjunto $\mathcal{A}_k$ (y, por tanto, en $\mathcal{L} \cup \mathcal{U}$), si $(i, k) \in \mathcal{L}$, $(k, j) \in \mathcal{U}$ e $(i, j) \notin \mathcal{A}_{k-1}$. Este nuevo elemento se llama *llenado*, aunque este término también se utiliza indistintamente para referirse al conjunto $(\mathcal{L} \cup \mathcal{U}) - \mathcal{A}$, es decir, para referirse al conjunto de las posiciones correspondientes a elementos no nulos en los factores, asociados a elementos nulos en A. Los algoritmos de factorización LU para matrices dispersas explotan dos observaciones con el objetivo de reducir el coste computacional y de almacenamiento:

1. No es necesario realizar la actualización si $(i, k) \notin \mathcal{L}$ o $(k, j) \notin \mathcal{U}$.

2. No es necesario calcular ni almacenar el elemento l_{ij}, si $(i, j) \notin \mathcal{A}$, e $(i, k) \notin \mathcal{L}$ o $(k, j) \notin \mathcal{U}$, para $k = 1, 2, \ldots, i - 1$.

Algoritmo 2.1: Variante algorítmica *right-looking* o KIJ de la factorización LU.

```
1  A_0 ← A
2  para k = 1 : n − 1 hacer
3      l_ik ← a_ik^(k−1) / a_kk^(k−1) con i = k + 1 : n
4      u_kj ← a_kj^(k−1) con j = k : n
5      para i = k + 1 : n hacer
6          para j = k + 1 : n hacer
7              a_ij^(k) ← a_ij^(k−1) − l_ik u_kj
8          fin para
9      fin para
10 fin para
```

La factorización LU se convierte en aproximada (o incompleta), es decir, en factorización ILU, cuando se descartan elementos de las matrices involucradas en el proceso de eliminación. Las distintas técnicas básicas para derivar precondicionadores ILU se diferencian esencialmente en la estrategia empleada para descartar elementos. Por ejemplo, se pueden descartar aquellos elementos situados en posiciones que no pertenecen a un patrón estático (preestablecido) de posiciones $\mathcal{P}$. El patrón $\mathcal{P}$ debe incluir las posiciones correspondientes a los elementos situados en la diagonal principal ($\{(1, 1), (2, 2), \ldots, (n, n)\}$), ya que en caso contrario el proceso de eliminación falla debido a divisiones por cero.

A continuación consideramos cómo se incorpora el descarte de elementos al proceso de eliminación Gaussiana, y derivamos la relación entre los factores que se obtienen al completar este proceso y la matriz de coeficientes del sistema. Para ello asumimos que el descarte de elementos se efectúa usando un patrón de descarte preestablecido $\mathcal{P}$, aunque las mismas conclusiones se pueden extraer empleando cualquier otro criterio.

En la iteración k-ésima, el proceso de eliminación (aproximada) parte de una aproximación a A_{k-1}, a la que llamamos $\hat{A}_{k-1}$. Posteriormente, se aplica una transformación Gaussiana a $\hat{A}_{k-1}$, es decir, $\tilde{A}_k = \tilde{L}_k^{-1} \hat{A}_{k-1}$, y, para limitar el número de elementos no nulos de $\tilde{A}_k$, se descartan ciertos elementos de $\tilde{A}_k$ como, *por ejemplo*, aquellos situados en posiciones que no pertenecen a un patrón estático (preestablecido) de posiciones $\mathcal{P}$. Si $\hat{A}_k$ es la matriz resultante de descartar

elementos de $\tilde{A}_k$, entonces

$$\hat{A}_k = \tilde{A}_k - R_k, \tag{2.7}$$

con $R_k = (r_{ij}^{(k)})$ definidos de la siguiente forma

$$r_{ij}^{(k)} = \begin{cases} 0 & si \quad 1 \leq i \leq k \text{ o } 1 \leq j \leq k\,, \\ 0 & si \quad k < i \leq n,\ k < j \leq n \text{ e } (i,j) \in \mathcal{P}\,, \\ \tilde{a}_{ij}^{(k)} & si \quad k < i \leq n,\ k < j \leq n \text{ e } (i,j) \notin \mathcal{P}\,. \end{cases} \tag{2.8}$$

Las primeras k filas y columnas de R_k son nulas porque la transformación Gaussiana $\tilde{A}_k = \tilde{L}_k^{-1}\hat{A}_{k-1}$ únicamente modifica (actualiza) elementos del bloque resultante de la intersección de las $n-k$ últimas filas y columnas. Por lo tanto, los elementos de R_k correspondientes a este bloque son nulos o iguales a $\tilde{a}_{ij}^{(k)}$, en función de si (i,j) pertenece o no a $\mathcal{P}$, respectivamente.

Teniendo en cuenta las consideraciones del párrafo anterior, es posible derivar la relación que existe entre $\tilde{A}_0 = A_0 = A$ y $\hat{A}_{n-1}$. Esta última matriz se obtiene a través de la recurrencia

$$\hat{A}_j = \tilde{L}_j^{-1}\hat{A}_{j-1} - R_j, \tag{2.9}$$

con $j = 1, 2, \ldots, n-1$. Por lo tanto, combinando los términos correspondientes a la misma, se tiene que

$$\tilde{U} = \hat{A}_{n-1} = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_1^{-1}\hat{A}_0 - \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_2^{-1}R_1 - \cdots - \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{k+1}^{-1}R_k - \cdots - R_{n-1}, \tag{2.10}$$

es una matriz triangular superior. Para $s = 1, 2, \ldots, n-1$, las primeras s filas y columnas de R_s son nulas y, por tanto, se cumple que $\tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{s+1}^{-1}R_s = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_{s+1}^{-1}\tilde{L}_s^{-1}\cdots\tilde{L}_1^{-1}R_s$. Teniendo en cuenta, además, que $\hat{A}_0 = \tilde{A}_0 - R_0$, entonces (2.10) puede simplificarse como

$$\tilde{U} = \tilde{L}_{n-1}^{-1}\cdots\tilde{L}_1^{-1}(A - (R_0 + \cdots + R_{n-1})) = \tilde{L}^{-1}(A - R). \tag{2.11}$$

Multiplicando ambas partes de la igualdad por $\tilde{L}$ y despejando A obtenemos que

$$A = \tilde{L}\tilde{U} + R, \tag{2.12}$$

donde $\tilde{L} = \tilde{L}_1\cdots\tilde{L}_{n-1}$ es una matriz triangular inferior unidad, $\tilde{U} = \hat{A}_{n-1}$ es una matriz triangular superior, y $R = \sum_{k=0}^{n-1} R_k$ es la matriz residuo o matriz de error. Si denotamos a los elementos de la matriz de error como $R = (r_{ij})$, entonces $r_{ij} = 0$ si $(i,j) \in \mathcal{P}$, y $r_{ij} = \sum_{k=0}^{\min(i,j)-1} \tilde{a}_{ij}^{(k)}$ si $(i,j) \notin \mathcal{P}$. La expresión (2.12) es independiente del criterio que se emplee para descartar elementos, dado que este criterio únicamente influye sobre la caracterización de la matriz de error R. Por tanto, podemos decir que (2.12) es la expresión general de la factorización ILU.

Factorización ILU$_\mathcal{P}$

La factorización ILU que descarta aquellos elementos que no pertenecen a un patrón preestablecido de posiciones se conoce como factorización ILU$_\mathcal{P}$. La variante algorítmica *right-looking* de esta factorización se muestra en el Algoritmo 2.2. La línea 1 del algoritmo inicializa la matriz $\hat{A}_0$ con los elementos de A que pertenecen a $\mathcal{P}$. Cada iteración del bucle externo obtiene la parte triangular inferior estricta de la columna k-ésima de $\tilde{L}$ (línea 3) y la parte superior de la fila k-ésima de $\tilde{U}$ (línea 4). Los dos bucles internos anidados calculan $\hat{A}_k$ de forma acorde a (2.9). El bucle de la línea 5 recorre *únicamente* aquellos elementos de la parte triangular inferior estricta de la columna k-ésima que pertenecen a $\mathcal{P}$; análogamente, el bucle de la línea 6, recorre *únicamente* aquellos de la parte triangular superior de la fila k-ésima que pertenecen a

$\mathcal{P}$. Por lo tanto, ambos bucles *no recorren* aquellas combinaciones de i y de j para las cuales está garantizado que $l_{ik}u_{kj} = 0$, evitando calcular de esa manera operaciones innecesarias al obtener $\tilde{L}_k^{-1}\hat{A}_{k-1}$. Además, el algoritmo ahorra operaciones adicionales evitando efectuar las actualizaciones correspondientes a posiciones (i, j) que no pertenecen a $\mathcal{P}$ (línea 7), y también ahorra memoria, ya que sólo es necesario almacenar los elementos que pertenecen a $\mathcal{P}$.

Algoritmo 2.2: Variante algorítmica *right-looking* o KIJ de la factorización ILU$_\mathcal{P}$.

```
1  â_ij^(0) ← a_ij con (i,j) ∈ P
2  para k = 1 : n − 1 hacer
3      l̃_ik ← â_ik^(k−1) / â_kk^(k−1) con i = k + 1 : n, (i,k) ∈ P
4      ũ_kj ← â_kj^(k−1) con j = k : n, (k,j) ∈ P
5      para i = k + 1 : n, (i,k) ∈ P hacer
6          para j = k : n, (k,j) ∈ P hacer
7              â_ij^(k) ← â_ij^(k−1) − l̃_ik ũ_kj si (i,j) ∈ P
8          fin para
9      fin para
10 fin para
```

La variante *right-looking* de la factorización ILU goza de interés teórico, ya que ésta se puede derivar de forma inmediata a partir del proceso de eliminación Gaussiana. Sin embargo, desde el punto de vista práctico, es complejo implementar de forma eficiente la actualización correspondiente a los dos bucles internos, teniendo en cuenta que las matrices dispersas habitualmente se almacenan según el formato CSR o CSC (consultar el Apéndice A). En el caso de la factorización ILU$_\mathcal{P}$, el inconveniente es menos severo, ya que el patrón de elementos no nulos de los factores se conoce de antemano y, por tanto, si las estructuras de datos empleadas para almacenarlos se crean en una etapa previa a la factorización, no es necesario acomodar dinámicamente nuevos elementos durante el proceso. Sin embargo, cuando la estructura de los factores triangulares no se conoce de antemano, la solución a este problema se gestiona habitualmente cambiando las estructuras de datos y/o la variante algorítmica de la factorización ILU. Para acomodar el formato de almacenamiento CSR (o CSC), a menudo se opta por una variante algorítmica de la factorización diferente a la *right-looking*. En particular, en la práctica se utilizan variantes algorítmicas de actualización retardada (*delayed-update LU*). En este tipo de variantes, como por ejemplo, la *row-lazy* [84] (IKJ), la *column-lazy* [84] (JKI), o la *row-column-lazy* [84] (variante de Crout), todas las actualizaciones de una iteración del bucle externo se efectúan, respectivamente, sobre una única fila, columna, o fila y columna de tal forma que, una vez que se han calculado, es posible incorporarlas fácilmente a las estructuras de datos empleadas para los factores. A continuación revisamos la variante algorítmica IKJ de la factorización ILU$_\mathcal{P}$.

La variante algorítmica IKJ de la factorización ILU$_\mathcal{P}$ se muestra en el Algoritmo 2.3. La iteración i-ésima del bucle externo obtiene los elementos de la fila i-ésima situados en la parte triangular inferior estricta de $\tilde{L}$ (línea 5) y los de la fila i-ésima situados en la parte triangular superior de $\tilde{U}$ (línea 10). Por esta razón, CSR constituye el formato de almacenamiento más apropiado para las estructuras de datos correspondientes a $\tilde{L}$ y a $\tilde{U}$. Para cada valor de k, el bucle de la línea 4 elimina (anula) un nuevo elemento $\hat{a}_{ik}^{(k-1)}$ de la fila i-ésima de $\hat{A}_{k-1}$ si $(i, k) \in \mathcal{P}$. Para ello, el bucle de la línea 6 resta a la fila i-ésima de $\hat{A}_{k-1}$ el resultado de multiplicar los elementos de fila k-ésima situados en la parte triangular superior $\hat{A}_{k-1}$ (es decir, la parte triangular superior de $\tilde{U}$) por $\tilde{l}_{ik}$. Como resultado de esta eliminación se obtiene la fila i-ésima de $\hat{A}_k$; en la Figura 2.1 se muestran los elementos de las matrices $\hat{A}_{k-1}$ y $\hat{A}_k$ involucrados en la actualización que efectúa el bucle de la línea 6 con $i = 5$, $k = 2$, y un patrón $\mathcal{P}$ de ejemplo. Al finalizar el bucle de la línea 4, todos los elementos de la parte triangular estricta de la fila i-ésima

que pertenecen a $\mathcal{P}$ ya se han eliminado y, por tanto, se obtiene la fila i-ésima de $\hat{A}_{i-1}$, es decir, de $\tilde{U}$. Las implementaciones eficientes de la variante IKJ de la factorización ILU$_\mathcal{P}$ no generan explícitamente las matrices $\hat{A}_k$, con $k = 0, 1, \ldots, n-1$. En su lugar, todas las actualizaciones correspondientes al bucle k se efectúan sobre un vector de trabajo w, que además almacena los elementos de la fila i-ésima de $\tilde{L}$ en las posiciones correspondientes a elementos ya eliminados. En el Algoritmo 2.4 se recoge la variante IKJ cuando se emplea este vector de trabajo.

La variante IKJ y la variante *right-looking* de la factorización ILU$_\mathcal{P}$ producen factores incompletos equivalentes para el mismo patrón $\mathcal{P}$, si éste se preestablece y no sufre cambios durante la factorización [138]. Por el contrario, si $\mathcal{P}$ se genera dinámicamente durante el proceso de eliminación, entonces no existen garantías de que ambas variantes produzcan el mismo resultado.

Algoritmo 2.3: Variante algorítmica *row-lazy* o IKJ de la factorización ILU$_\mathcal{P}$.

```
1  â_ij^(0) ← a_ij con (i,j) ∈ P
2  ũ_1j ← â_1j^(0) con j = 1:n, (1,j) ∈ P
3  para i = 2:n hacer
4      para k = 1:i−1, (i,k) ∈ P hacer
5          l̃_ik ← â_ik^(k−1) / â_kk^(k−1)
6          para j = k+1:n, (k,j) ∈ P hacer
7              â_ij^(k) ← â_ij^(k−1) − l̃_ik ũ_kj si (i,j) ∈ P
8          fin para
9      fin para
10     ũ_ij ← â_ij^(i−1) con j = i:n, (i,j) ∈ P
11 fin para
```

Algoritmo 2.4: Implementación eficiente de la variante *row-lazy* o IKJ de la factorización ILU$_\mathcal{P}$.

```
1  para i = 1:n hacer
2      w_j ← a_ij con j = 1:n, (i,j) ∈ P
3      para k = 1:i−1, (i,k) ∈ P hacer
4          w_k ← w_k / ũ_kk
5          para j = k+1:n, (k,j) ∈ P hacer
6              w_j ← w_j − w_k ũ_kj si (i,j) ∈ P
7          fin para
8      fin para
9      l̃_ij ← w_j y w_j ← 0 con j = 1:i−1, (i,j) ∈ P
10     ũ_ij ← w_j y w_j ← 0 con j = i:n, (i,j) ∈ P
11 fin para
```

Factorización ILU(0) y su generalización ILU(l) o de nivel de llenado

Si en la factorización ILU$_\mathcal{P}$, $\mathcal{P}$ se toma igual a $\mathcal{A}$, entonces obtenemos la factorización ILU sin llenado, también llamada factorización ILU(0). En este caso los patrones de elementos no nulos de $\tilde{L}$ y $\tilde{U}$ coinciden con el patrón de elementos no nulos de la parte triangular inferior y superior de A, respectivamente. Es decir, $(\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}) - \mathcal{A} = \emptyset$. Además, $A = \tilde{L}\tilde{U}$ para aquellos elementos situados en las posiciones que pertenecen a $\mathcal{A}$. El mismo concepto se puede aplicar para la factorización de Cholesky $A = LL^T$ en el caso de matrices SDP, obteniendo en este caso el precondicionador IC (*Incomplete Cholesky*) sin llenado, o análogamente IC(0). Los precondicionadores sin llenado

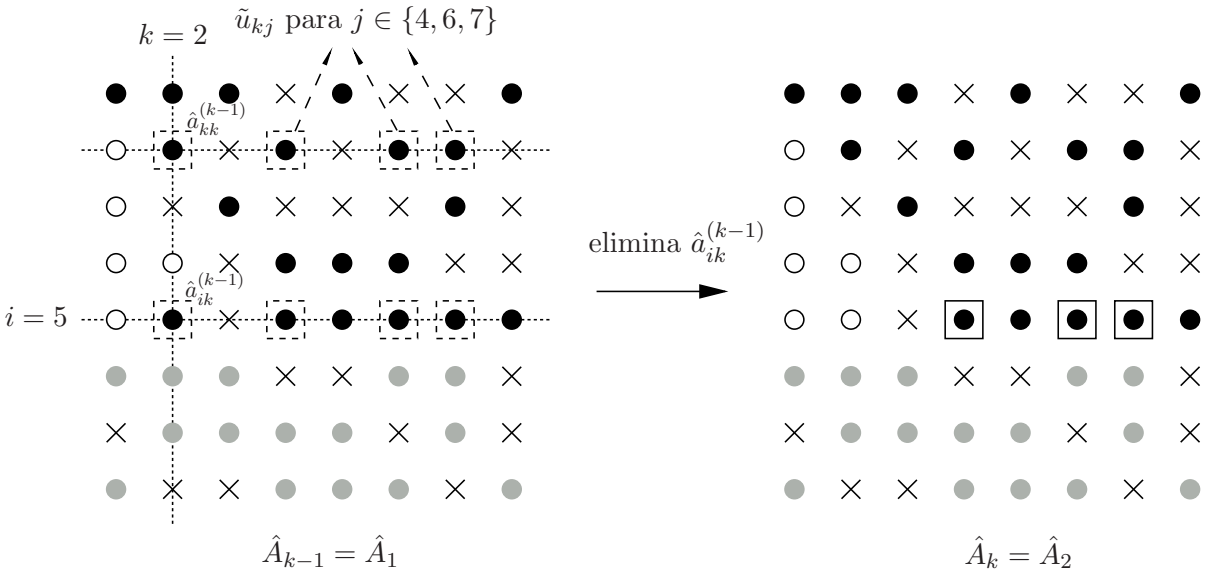


Figura 2.1: Elementos de las matrices $\hat{A}_{k-1}$ (izquierda) y $\hat{A}_k$ (derecha) involucrados en la actualización que efectúa el bucle de la línea 6 de la variante IKJ de la factorización $\text{ILU}_{\mathcal{P}}$ (ver Algoritmo 2.3) con $i = 5$, $k = 2$, y un patrón $\mathcal{P}$ de ejemplo. Las aspas corresponden a elementos que están situados en posiciones que no pertenecen a $\mathcal{P}$, y los círculos a elementos que sí lo están. Los círculos de color gris indican elementos que no han sido accedidos todavía por el algoritmo, los de color blanco son elementos que ya han sido eliminados, y los de color negro elementos que ya han sido calculados en iteraciones previas ($i = 1, \ldots, 4$), o que se calculan en la iteración actual ($i = 5$) del bucle externo. Los elementos recuadrados con línea continua o discontinua son aquellos accedidos por el algoritmo en modo de escritura o de lectura, respectivamente.

son fáciles de implementar, su coste es relativamente pequeño, y además son bastante eficaces en problemas significativos, como por ejemplo en sistemas que surgen en la resolución numérica de EDPs elípticas mediante el método de las diferencias finitas. De hecho las factorizaciones ILU sin llenado se propusieron originalmente para este tipo de problemas (consultar, por ejemplo, [138]). Sin embargo, para problemas más complejos, las factorizaciones ILU sin llenado obtienen con frecuencia aproximaciones de A muy poco precisas, es decir, con $\|R\|$ "grande" en (2.12). La causa fundamental es el descarte de muchos elementos de gran tamaño en el proceso de factorización.

Para intentar aumentar la precisión de la factorización aproximada se pueden incluir posiciones en $\mathcal{P}$ no presentes en el patrón de elementos no nulos de A. En la familia de precondicionadores de *nivel de llenado* l o ILU(l), estas posiciones se escogen dentro del conjunto $\mathcal{L}\cup\mathcal{U}$, es decir, dentro del conjunto de posiciones en las que se sitúan los elementos no nulos de $L+U$. Por lo tanto, para esta familia de precondicionadores se cumple que $\mathcal{P}=\tilde{\mathcal{L}}\cup\tilde{\mathcal{U}}\subseteq\mathcal{L}\cup\mathcal{U}$. El parámetro l es un número entero preestablecido que determina qué posiciones del conjunto $\mathcal{L}\cup\mathcal{U}$ formarán parte del patrón $\mathcal{P}$. Este patrón se construye en una etapa previa a la factorización numérica, conocida como factorización simbólica ILU(l), que queda recogido en el Algoritmo 2.5. En el proceso se tienen en cuenta las posiciones de los elementos involucrados en la eliminación, y no sus valores numéricos. El algoritmo asigna un número entero $\bar{p}_{ij}$, denominado *nivel de llenado*, a cada posición (i,j) "aceptada", es decir, a cada posición que se elige como integrante de $\mathcal{P}$. En la iteración i-ésima, todas las posiciones (i,j) de la fila i-ésima que pertenecen al patrón de elementos no nulos de A se aceptan, y el algoritmo les asigna nivel de llenado $\bar{p}_{ij}=0$ (línea 3). Posteriormente, el bucle de la línea 4 procede con la eliminación de las posiciones (i,k), con $k<i$, que pertenecen a $\mathcal{P}$ en *orden ascendente*[1]. Al eliminar una posición concreta (i,h), cualquiera de las posiciones (h,j) recorridas por el bucle de la línea 5 *puede causar* llenado en la posición (i,j). El nivel de llenado w correspondiente a la posición (i,j) se obtiene como la suma del nivel de llenado de las entradas causantes más 1 (línea 6). Si el valor de w es igual o inferior al de la constante prefijada l, entonces la posición (i,j) se acepta en la línea 11. El llenado de una posición (i,j) puede estar causado por varios pares de entradas (i,k), $(k,j)\in\mathcal{P}$, con $1\leq k<\text{mín}(i,j)$. La factorización ILU(l) simbólica asigna a las posiciones de $\mathcal{P}$ el nivel de llenado más pequeño posible (línea 9). Nótese que si $l=0$, entonces el algoritmo obtiene $\mathcal{P}=\mathcal{A}$, es decir, el patrón correspondiente al precondicionador ILU(0), mientras que si $l=\infty$, entonces $\mathcal{P}=\mathcal{L}\cup\mathcal{U}$, es decir, el correspondiente a la factorización LU de A. En la Figura 2.2 se muestra el patrón $\mathcal{P}$ calculado por la factorización simbólica ILU(l) para una matriz dispersa de ejemplo y cuatro valores distintos de l.

El Algoritmo 2.5 está respaldado por el *teorema del camino de llenado incompleto* y su relación con el *teorema del camino de llenado* [97]. Estos resultados caracterizan la presencia de una posición (i,j) en $\mathcal{P}$ y en $\mathcal{L}\cup\mathcal{U}$, respectivamente, en función de los *caminos de llenado* del grafo de adyacencia $G(A)$. Un camino de llenado en $G(A)$ entre los vértices con identificadores i y j es una secuencia de aristas de longitud superior o igual a uno que los conecta a través de vértices con identificador inferior a $\text{mín}(i,j)$. El teorema del camino de llenado establece que $(i,j)\in\mathcal{L}\cup\mathcal{U}$ si y solo si existe un camino de llenado de i a j en $G(A)$, mientras que el teorema del camino de llenado incompleto establece que la posición (i,j) pertenece a $\mathcal{P}$ con nivel de llenado $\bar{p}_{ij}=k$, si y solo si existe al menos un camino de llenado de i a j, y además el de longitud mínima entre todos los caminos de llenado de i a j es de longitud $k+1$. Por lo tanto, $\mathcal{P}\subseteq(\mathcal{L}\cup\mathcal{U})$. Además, el conjunto $(\mathcal{L}\cup\mathcal{U})-\mathcal{P}$ está formado por las posiciones (i,j) para las cuales todos los caminos de llenado de i a j tienen longitud superior a $l+1$. Por lo tanto, si medimos la distancia entre dos vértices en $G(A)$ en términos de la longitud de los caminos de

[1]En los Algoritmos 2.3-2.4 también era necesario eliminar los elementos en orden ascendente. Sin embargo, satisfacer este requisito en el caso del Algoritmo 2.5 puede incrementar significativamente el coste ya que el patrón $\mathcal{P}$ puede ser distinto para dos iteraciones diferentes del bucle de la línea 4 y, por lo tanto, es necesario mantener el orden en la estructura de datos usada para $\mathcal{P}$. En general, es necesario mantener este orden para todos los algoritmos de factorización basados en la variante IKJ que generan dinámicamente la estructura de cada una de las filas de los factores.

Algoritmo 2.5: Factorización simbólica ILU(l).

```
1  P ← ∅
2  para i = 1 : n hacer
3      insertar (P, (i, j)) y p̄_ij ← 0 con (i, j) ∈ A
4      para k = 1 : i − 1 en orden ascendente, (i, k) ∈ P hacer
5          para j = k + 1 : n, (k, j) ∈ P hacer
6              w ← p̄_ik + p̄_kj + 1
7              si w ≤ l entonces
8                  si (i, j) ∈ P entonces
9                      p̄_ij ← mín(p̄_ij, w)
10                 si no
11                     insertar (P, (i, j))
12                     p̄_ij ← w
13                 fin si
14             fin si
15         fin para
16     fin para
17 fin para
```

llenado, en el precondicionador ILU(l) se aceptan aquellas posiciones (i, j) para las cuales los vértices i y j están "cerca" en $G(A)$.

En muchos casos la eficacia del precondicionador ILU(l) en términos de su capacidad para acelerar la convergencia del método iterativo, mejora significativamente considerando niveles de llenado adicionales, es decir, valores de l mayores. Sin embargo, cuando $l \geq 1$ es posible que $\mathcal{P}$ pueda acercarse mucho a $\mathcal{L} \cup \mathcal{U}$, incluso con valores relativamente pequeños de l. Aunque en la Figura 2.2 no se observa este comportamiento para $l \leq 2$, un número considerable de ejemplos de la colección de matrices dispersas Harwell-Boeing tienen niveles "máximos" de llenado relativamente pequeños. El nivel máximo de llenado es aquel nivel de llenado l más pequeño para que se cumple que la factorización ILU(l) coincide con la LU. Si en estos casos es necesario hacer uso de valores de $l \geq 1$ para obtener precondicionadores eficaces, entonces el resolutor compuesto por el binomio precondicionador ILU(l)-método iterativo deja de ser eficiente comparado con los métodos directos de resolución. Otro problema adicional es que, excepto en casos muy concretos, no es posible predecir el coste computacional y de almacenamiento requerido por los precondicionadores ILU(l). Sin embargo, el inconveniente principal de los precondicionadores de nivel de llenado es que, para muchas matrices, especialmente para matrices indefinidas (es decir, con valores propios positivos y negativos) lejos de ser diagonal dominantes, el nivel de llenado puede no ser un buen indicador de la magnitud de los elementos descartados. Por lo tanto, el algoritmo puede descartar muchos elementos de gran magnitud, obteniendo en este caso una factorización ILU de poca precisión, es decir, con $\|R\|$ "grande". Aunque no existe una relación *directa* entre $\|R\|$ y la eficacia del precondicionador, la experiencia revela que, habitualmente, este factor causa un incremento del número de iteraciones requeridas por el método iterativo para resolver el sistema.

Descarte de elementos en base a un umbral de magnitud: factorización ILUT(τ, p)

Como alternativa a las técnicas de descarte basadas en la posición, los elementos pueden ser descartados durante la factorización en base a su magnitud, en un intento *voraz* de mejorar la eficacia del precondicionador minimizando $\|R\|$ en (2.12). La ILUT(τ, p) [137] (*ILU with Threshold*) es la factorización aproximada de referencia con descarte de elementos en base a su magnitud. También es posible definir la versión correspondiente para matrices SDP o simétricas

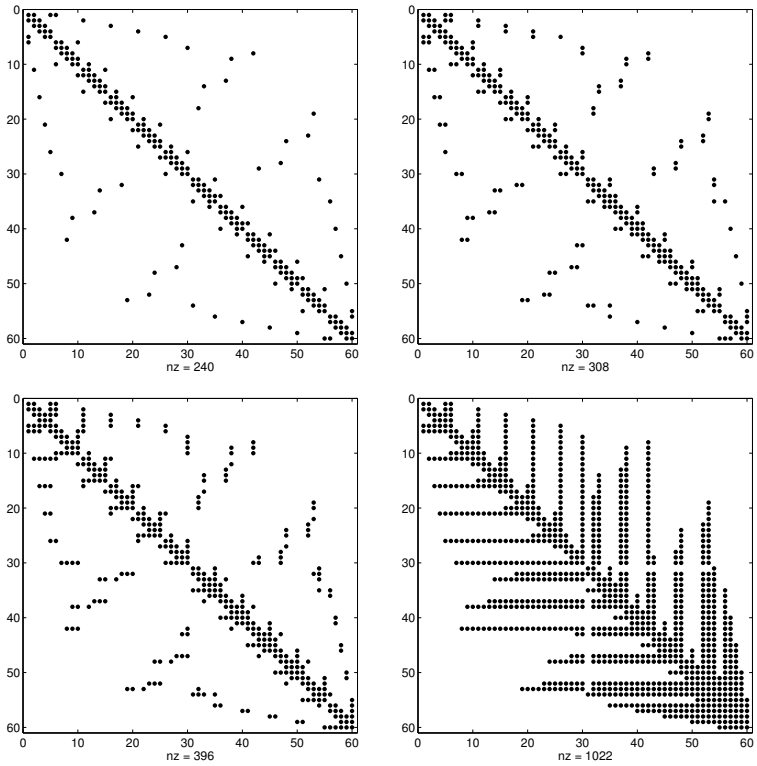


Figura 2.2: Patrón $\mathcal{P}$ calculado por la factorización simbólica ILU(l) para un matriz dispersa de ejemplo y cuatro valores distintos de l. De izquierda a derecha y de arriba a abajo: $l = 0, 1, 2, \infty$. El patrón de elementos no nulos de A coincide con el patrón de la factorización ILU(0).

indefinidas, obteniendo el precondicionador ICT(τ, p) [25] (*Incomplete Cholesky with Threshold*) o el SILUT(τ, p) [25] (*Symmetric ILU with Threshold*), respectivamente. Estas variantes están motivadas por el hecho de que en la factorización ILUT no existen garantías de que $M = \tilde{L}\tilde{U}$ sea una matriz simétrica aún siéndolo A, además de por el ahorro de coste computacional y de almacenamiento que supone explotar la simetría de A. La siguiente discusión se ciñe a la variante de la factorización ILUT para matrices generales.

La factorización ILUT(τ, p) se muestra en el Algoritmo 2.6. El parámetro τ es un número real positivo, denominado *tolerancia de descarte*, que controla la magnitud de los elementos admitidos en $\tilde{L}$ y en $\tilde{U}$, y el parámetro p es un valor entero positivo que controla el número máximo de elementos admitidos en cada fila de $\tilde{L}$ y de $\tilde{U}$. Como en el caso de los Algoritmos 2.3-2.4, está basada en la variante IKJ de la factorización ILU. El bucle de la línea 4 únicamente elimina los elementos w_k ($\hat{a}_{ik}^{(k-1)}$) de la fila i-ésima situados en la parte triangular inferior estricta que son nulos. Un elemento $w_k \neq 0$ se descarta (línea 7) si la magnitud de $w_k/\tilde{u}_{kk}$ es menor o igual al umbral τ_i, ahorrando de esa forma las actualizaciones que conlleva su eliminación. En caso contrario, w_k se elimina de forma acorde a los Algoritmos 2.3-2.4, aunque únicamente se actualizan aquellos elementos w_j para los cuales $\tilde{u}_{kj} \neq 0$ y, además, el llenado causado por las actualizaciones de la línea 10 se acepta *temporalmente*. Tras finalizar el bucle de la línea 4, los elementos de la parte triangular superior de w de magnitud inferior a τ_i se descartan en la línea 14. Posteriormente, en la línea 15 (y 16) se descartan los elementos de magnitud inferior a los p elementos de mayor magnitud de w situados en la parte triangular inferior estricta (y superior). De esta forma, $|\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}| \leq 2np$. Por último, en la línea 17 (18) se añade la fila i-ésima a la estructura de datos en la que se almacena $\tilde{L}$ ($\tilde{U}$).

El Algoritmo 2.6 emplea una tolerancia de descarte *relativa* a la magnitud de la fila i-ésima de A para descartar los elementos correspondientes a la fila i-ésima de $\tilde{L}$ (línea 7) y de $\tilde{U}$

Algoritmo 2.6: Factorización ILUT(τ, p)

```
1  para i = 1 : n hacer
2      τ_i ← τ‖e_i^T A‖
3      w_j ← a_ij con j = 1 : n, a_ij ≠ 0
4      para k = 1 : i − 1 en orden ascendente, w_k ≠ 0 hacer
5          w_k ← w_k/ũ_kk
6          si |w_k| ≤ τ_i entonces
7              w_k ← 0
8          si no
9              para j = k : n, ũ_kj ≠ 0 hacer
10                 w_j ← w_j − w_k ũ_kj
11             fin para
12         fin si
13     fin para
14     w_j ← 0 con j = i : n, |w_j| ≤ τ_i
15     Sesgar la parte triangular inferior estricta de w a p elementos
16     Sesgar la parte triangular superior estricta de w a p elementos
17     l̃_ij ← w_j con j = 1 : i − 1, w_j ≠ 0
18     ũ_ij ← w_j con j = i : n, w_j ≠ 0
19     w ← 0
20 fin para
```

(línea 14). Esta tolerancia relativa, τ_i, se obtiene en la línea 2 como el producto de la tolerancia de descarte *absoluta*, τ, y la norma de la fila i-ésima de A, $\|e_i^T A\|$. También es posible definir la tolerancia relativa para los elementos de la fila i-ésima de $\tilde{U}$ considerando $|\tilde{u}_{ii}|$ o $\|e_i^T \tilde{U}\|$, en lugar de $\|e_i^T A\|$. Los criterios de descarte basados en tolerancias relativas son, en general, más fiables que el criterio basado exclusivamente en la tolerancia absoluta, y resultan con frecuencia una buena solución de compromiso. Un inconveniente común a todos estos criterios es la elección de un valor satisfactorio para τ. Para esta selección, habitualmente se parte de un subconjunto representativo de los sistemas lineales a resolver en el contexto de una aplicación concreta y, mediante experimentación, se determina un valor apropiado para los sistemas que surgen de la misma. En muchos casos se obtienen buenos resultados seleccionando valores de τ en el intervalo $[10^{-4}, 10^{-2}]$, aunque el óptimo depende en gran medida de cada problema concreto [137].

En el proceso de factorización ILUT, los elementos w_k que se descartan en la línea 7 *no se eliminan*, ahorrando de esa forma actualizaciones del bucle de la línea 9. Nótese la diferencia entre eliminar y descartar: para eliminar un elemento w_k es necesario efectuar las actualizaciones correspondientes al bucle de la línea 9. Posteriormente, w_k pasa a almacenar el valor $w_k/\tilde{u}_{kk}$, es decir, la entrada correspondiente a $\tilde{L}$. Sin embargo, cuando un elemento w_k se descarta, éste se substituye "artificialmente" por el valor cero. Una posible modificación del Algoritmo 2.6 podría eliminar todos los elementos no nulos de la parte triangular inferior estricta y, posteriormente, descartar los elementos de $\tilde{L}$ de magnitud inferior a τ_i en la línea 14, de forma análoga a $\tilde{U}$. Sin embargo, es posible demostrar que esta modificación de la factorización ILUT no sólo incrementa el coste computacional, sino que además *disminuye la precisión de la aproximación* [50]. Por lo tanto, existe una justificación matemática para no eliminar los elementos descartados, o lo que es equivalente, para no descartar (a posteriori) los elementos de $\tilde{L}$ que se han empleado en el proceso de eliminación. Es por ello que, desde el punto de vista matemático, sesgar las filas de $\tilde{L}$ no se considera una buena práctica, aunque ello suponga no disponer a priori de un umbral para $|\tilde{\mathcal{L}} \cup \tilde{\mathcal{U}}|$. De nuevo, la combinación óptima para los valores de p y τ depende en gran medida del problema concreto, aunque habitualmente se obtienen buenos resultados ajustando únicamente el valor de τ y fijando p a un valor "grande" [110]. Por ejemplo, el valor por defecto para p en

el paquete ILUPACK [35] es $n+1$.

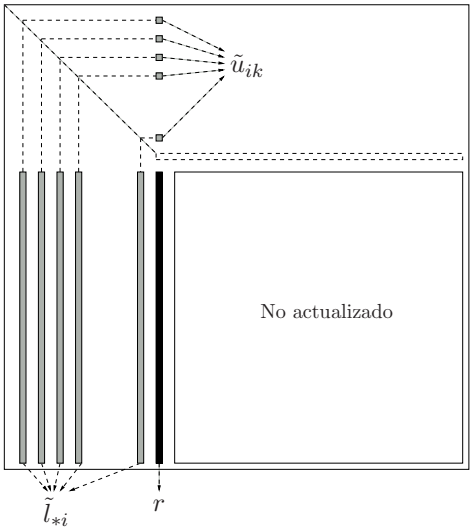
En las implementaciones eficientes del Algoritmo 2.6, w es una estructura de datos, como por ejemplo, un montículo (*heap*) o un árbol binario de búsqueda, que mantiene el orden entre los identificadores de columna de los elementos *no nulos* a medida que se produce llenado en w. Operaciones del tipo "$w_k \leftarrow 0$" se implementan eliminando el elemento w_k de la estructura de datos; las del tipo "$w_j \leftarrow w_j - w_k\tilde{u}_{kj}$", actualizando o insertando un nuevo elemento si la operación causa llenado; mientras que consultas del tipo "$w_k \neq 0$" vienen dadas de forma implícita al extraer el *siguiente* elemento de w, es decir, el elemento no nulo situado en la columna con identificador inmediatamente superior al de la iteración anterior del bucle de la línea 4. Cuando el llenado admitido por la factorización es relativamente pequeño, el coste requerido para mantener esta estructura no supone una parte significativa del coste global del proceso. Sin embargo, cuando se admite progresivamente más llenado (reduciendo el valor de τ y/o aumentando el de p), este coste puede incluso llegar a dominar el tiempo total de la factorización [138].

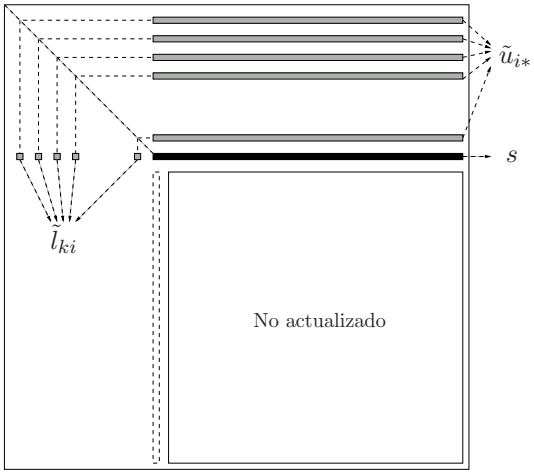
Variante algorítmica de Crout: factorización ILUC

El coste de mantener w en la variante IKJ de la factorización ILU se puede eliminar cambiando de variante. En particular, la variante de Crout [138] es una versión de *actualización retardada* de la factorización, y no requiere un orden en la eliminación de los elementos no nulos de la fila i-ésima, es decir, de los elementos $\hat{a}_{ik}^{(k-1)} \neq 0$ para $k = 1, \ldots, i-1$. En un iteración del bucle externo k de esta variante, las actualizaciones correspondientes a la eliminación de la columna k-ésima *se retardan*, es decir, se aplican en iteraciones posteriores. Además, al comienzo de la iteración k-ésima, las actualizaciones correspondientes a la eliminación de las $k-1$ primeras columnas *no se han aplicado todavía* sobre el bloque formado por la intersección de las últimas $n-k+1$ filas y columnas. Así pues, la iteración k-ésima de la variante de Crout aplica las actualizaciones retardadas en las $k-1$ iteraciones anteriores sobre la primera fila y columna de este bloque, es decir, sobre los elementos de la columna k-ésima situados en la parte triangular inferior estricta de A, y los elementos de la fila k-ésima situados en la parte triangular superior de A. En la parte superior de la Figura 2.3 se muestran los elementos involucrados en las actualizaciones que efectúa la iteración k-ésima de la factorización de Crout y, en la parte inferior, los pseudocódigos encargados de efectuarlas. La actualización de la columna k-ésima se muestra en la parte izquierda de la Figura 2.3 y, la de la fila k-ésima, en la parte derecha. Las filas y columnas de color gris contienen elementos de $\tilde{L}$ y de $\tilde{U}$ accedidos en modo lectura; la fila y columna de color negro contienen los elementos que se actualizan en la iteración k-ésima; r y s son dos vectores de trabajo; y

$$\tilde{l}_{*i} = \begin{pmatrix} \tilde{l}_{k+1,i} & \ldots & \tilde{l}_{n,i} \end{pmatrix}^T, \quad \tilde{u}_{i*} = \begin{pmatrix} \tilde{u}_{i,k} & \ldots & \tilde{u}_{i,n} \end{pmatrix}, \; i = 1 \ldots k-1. \tag{2.13}$$

En los pseudocódigos de la Figura 2.3, las actualizaciones de r y s se pueden efectuar *en cualquier orden*, lo que supone una ventaja significativa con respecto a la variante IKJ desde el punto de vista del coste de la factorización y de su implementación. De hecho, en la implementación eficiente de la factorización ILU de Crout, r y s son, esencialmente, dos vectores de trabajo densos [110]. La estructura dispersa del problema se explota ahorrando actualizaciones si $\tilde{u}_{ik} = 0$ o $\tilde{l}_{ji} = 0$ para el pseudocódigo de la parte izquierda de la Figura 2.3, o si $\tilde{l}_{ki} = 0$ o $\tilde{u}_{ij} = 0$ para el de la parte derecha. Además, las operaciones efectuadas por el pseudocódigo de la parte izquierda de la Figura 2.3 son *independientes* de las del pseudocódigo de la parte derecha y, por lo tanto, ambos códigos pueden ser ejecutados en paralelo. Aunque este tipo de paralelismo puede ser explotado para reducir el coste de la factorización en procesadores multihebra, los algoritmos paralelos escalables para calcular factorizaciones ILU explotan tipos de paralelismo en los que el grado de concurrencia es significativamente más alto. En el Apartado 2.3 se ofrecerán más detalles.





$r \leftarrow 0$
$r_i \leftarrow a_{ik}$ **con** $i = k+1 : n, a_{ik} \neq 0$
para $i = 1 : k-1, \tilde{u}_{ik} \neq 0$ **hacer**
  para $j = k+1 : n, \tilde{l}_{ji} \neq 0$ **hacer**
    $r_j \leftarrow r_j - \tilde{u}_{ik}\tilde{l}_{ji}$
  fin para
fin para

$s \leftarrow 0$
$s_j \leftarrow a_{kj}$ **con** $j = k : n, a_{kj} \neq 0$
para $i = 1 : k-1, \tilde{l}_{ki} \neq 0$ **hacer**
  para $j = k : n, \tilde{u}_{ij} \neq 0$ **hacer**
    $s_j \leftarrow s_j - \tilde{l}_{ki}\tilde{u}_{ij}$
  fin para
fin para

Figura 2.3: Actualizaciones efectuadas sobre la columna (izquierda) y fila (derecha) k-ésima de A en la iteración k-ésima de la variante de Crout de la factorización ILU.

Los pseudocódigos de la Figura 2.3 se completan aplicando descarte a los vectores de trabajo r y s, escalando los elementos de r por $(u_{kk})^{-1}$, añadiendo la columna k-ésima a $\tilde{L}$ y la fila k-ésima a $\tilde{U}$, e iterando para $k = 1, 2, \ldots, n$. La factorización ILUC (*Incomplete LU Crout*) se muestra en el Algoritmo 2.7. Los formatos de almacenamiento más apropiados para $\tilde{L}$ y $\tilde{U}$ son, respectivamente, CSC y CSR (consultar el Apéndice A), por lo que $\tilde{L}$ se almacena por columnas y $\tilde{U}$ por filas.

Desde el punto de vista de la implementación eficiente del Algoritmo 2.7, existen dos dificultades potenciales. La primera es el acceso a secciones parciales de las primeras $k-1$ filas y columnas de $\tilde{U}$ y de $\tilde{L}$, respectivamente; es decir, a $\tilde{l}_{*i}$ y $\tilde{u}_{i*}$ en (2.13). En el caso de $\tilde{U}$, este inconveniente se soluciona empleando un vector de trabajo de tamaño n, denominado `Ufirst`, que contiene para cada fila i, un puntero a la posición del "primer" elemento no nulo de la fila i-ésima situado en una columna con identificador superior o igual a k. Por "primer" elemento entendemos aquella entrada $\tilde{u}_{ij}$ que cumple que $j = \text{mín}(\{m : k \leq m \leq n, \tilde{u}_{im} \neq 0\})$. En el caso de $\tilde{L}$, se emplea un vector análogo denominado `Lfirst`. Ambos vectores se pueden actualizar fácilmente al final de cada iteración del bucle externo siempre y cuando, en la estructura de datos empleada para $\tilde{U}$ ($\tilde{L}$), los elementos de cada una de las filas (columnas) estén ordenados en orden creciente por su identificador de columna (fila). Esta ordenación se puede llevar a cabo en las líneas 18 y 19 del Algoritmo 2.7, cada vez que se genera una nueva columna y fila de los factores incompletos.

La segunda dificultad está asociada al recorrido de los elementos no nulos de la columna (fila) k-ésima de $\tilde{U}$ ($\tilde{L}$), teniendo en cuenta que $\tilde{U}$ ($\tilde{L}$) se almacena por filas (columnas). En [110] se presentó una solución eficiente y muy elegante a este problema. La idea clave es emplear una *lista enlazada* que contiene, en el caso de $\tilde{U}$, los identificadores de fila correspondientes a los elementos no nulos de la columna k-ésima de $\tilde{U}$. La lista se puede implementar mediante

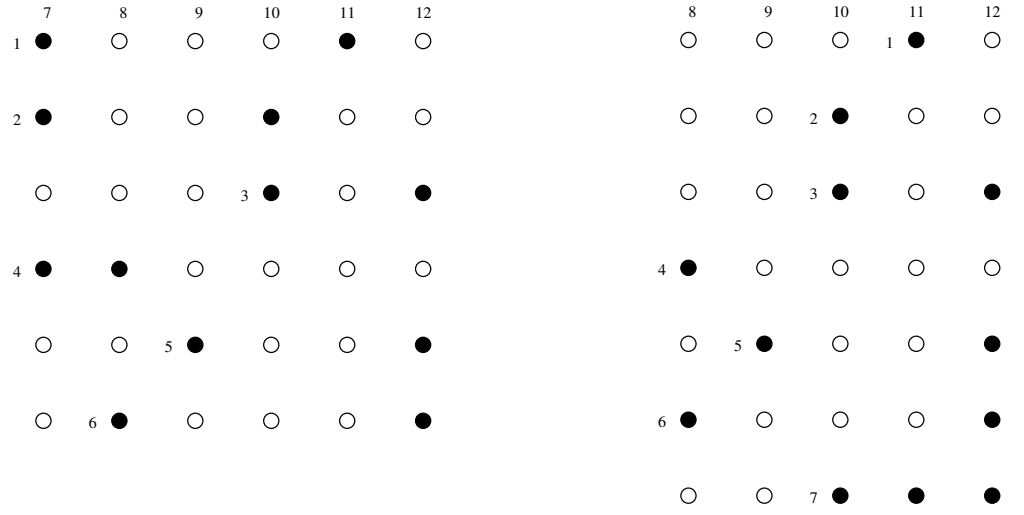
Algoritmo 2.7: Factorización ILUC

para $k = 1 : n$ **hacer**
  $r \leftarrow 0$
  $r_i \leftarrow a_{ik}$ **con** $i = k+1 : n, a_{ik} \neq 0$
  para $i = 1 : k-1, \tilde{u}_{ik} \neq 0$ **hacer**
    para $j = k+1 : n, \tilde{l}_{ji} \neq 0$ **hacer**
      $r_j \leftarrow r_j - \tilde{u}_{ik}\tilde{l}_{ji}$
    fin para
  fin para
  $s \leftarrow 0$
  $s_j \leftarrow a_{kj}$ **con** $j = k : n, a_{kj} \neq 0$
  para $i = 1 : k-1, \tilde{l}_{ki} \neq 0$ **hacer**
    para $j = k : n, \tilde{u}_{ij} \neq 0$ **hacer**
      $s_j \leftarrow s_j - \tilde{l}_{ki}\tilde{u}_{ij}$
    fin para
  fin para
  Aplicar descarte a r
  Aplicar descarte a s
  $\tilde{u}_{kj} \leftarrow s_j$ **con** $j = k : n, w_j \neq 0$
  $\tilde{l}_{jk} \leftarrow r_j/\tilde{u}_{kk}$ **con** $j = k+1 : n, w_j \neq 0$
fin para

un vector de trabajo `Ulist`, de tal forma que `Ulist(`k`)` almacena el identificador de fila del primer[2] elemento no nulo de la columna k de $\tilde{U}$, `Ulist(Ulist(`k`))` contiene el identificador de fila del segundo, y así sucesivamente hasta el último elemento de la lista. Si el último elemento de la lista pertenece a la fila j, entonces `Ulist(`j`)` tiene el valor 0. En la solución propuesta en [110], `Ulist` es un vector de tamaño n que contiene, además de la lista enlazada completa para la columna k, listas enlazadas parciales para las columnas con identificador superior a k. Al comienzo de la iteración k-ésima, la lista enlazada parcial de una determinada columna con identificador $j > k$ contiene aquellos identificadores de fila i para los cuales `Ufirst(`i`)` apunta a $\tilde{u}_{ij}$. Al final de la iteración k-ésima, se preparan las listas para la siguiente iteración. Para ello se recorren los identificadores de fila correspondientes a la lista enlazada de la columna k-ésima de $\tilde{U}$ y, por cada identificador i de esta lista, se actualiza `Ufirst(`i`)` para que apunte al "siguiente" elemento, es decir, al elemento $\tilde{u}_{is}$ que cumple que $s = \text{mín}(\{m : k+1 \leq m \leq n, \tilde{u}_{im} \neq 0\})$. Posteriormente, i se extrae de la lista enlazada de la columna k y se incorpora a la lista de la columna s. Por último, para terminar de preparar la iteración $k+1$, falta ajustar el puntero `Ufirst(`k`)` para que apunte al elemento $\tilde{u}_{kj}$, con $j = \text{mín}(\{m : k+1 \leq m \leq n, \tilde{u}_{km} \neq 0\})$, y añadir k a `Ulist(`j`)`. En la Figura 2.4 se muestra el estado de los vectores de trabajo `Ufirst` y `Ulist` al comienzo de la iteración $k = 7$ (izquierda) y de la iteración $k+1 = 8$ (derecha) para un factor $\tilde{U} \in \mathbb{R}^{12,12}$ de ejemplo.

La solución descrita en el párrafo anterior se utilizó de forma pionera en los primeros paquetes de software de métodos directos, aunque cayó en desuso a medida que creció el tamaño y la complejidad de las nuevas generaciones de los sistemas de ecuaciones. En los paquetes actuales de software de métodos directos se emplean técnicas que permiten gestionar de forma más eficiente la estructura dispersa del problema [56]. Sin embargo, en [110] se mostró que el uso de listas enlazadas para la factorización ILUC es una solución eficiente debido a que el llenado admitido

[2]Nótese que en la factorización ILUC las actualizaciones se pueden aplicar en cualquier orden. Por lo tanto, *en el contexto de las listas enlazadas*, el "primer" elemento puede ser cualquiera de los elementos no nulos de la columna k-ésima de $\tilde{U}$.



Ufirst	
posición	valor
1	$\&\tilde{u}_{1,7}$
2	$\&\tilde{u}_{2,7}$
3	$\&\tilde{u}_{3,10}$
4	$\&\tilde{u}_{4,7}$
5	$\&\tilde{u}_{5,9}$
6	$\&\tilde{u}_{6,8}$
7	0
8	0
9	0
10	0
11	0
12	0

Ulist	
posición	valor
1	0
2	4
3	0
4	1
5	0
6	0
7	2
8	6
9	5
10	3
11	0
12	0

Ufirst	
posición	valor
1	$\&\tilde{u}_{1,11}$
2	$\&\tilde{u}_{2,10}$
3	$\&\tilde{u}_{3,10}$
4	$\&\tilde{u}_{4,8}$
5	$\&\tilde{u}_{5,9}$
6	$\&\tilde{u}_{6,8}$
7	$\&\tilde{u}_{7,10}$
8	0
9	0
10	0
11	0
12	0

Ulist	
posición	valor
1	0
2	3
3	0
4	6
5	0
6	0
7	2
8	4
9	5
10	7
11	1
12	0

Figura 2.4: Estado de los vectores de trabajo `Ufirst` y `Ulist` al comienzo de la iteración $k = 7$ (izquierda) y de la iteración $k + 1 = 8$ (derecha) para un factor $\tilde{U} \in \mathbb{R}^{12,12}$ de ejemplo. Los círculos de color blanco representan elementos nulos y los de color negro elementos no nulos. Los identificadores que aparecen a la izquierda de alguno de los elementos no nulos de una columna, son los identificadores que forman parte de la lista enlazada correspondiente a la misma. Expresiones del tipo "$\&\tilde{u}_{ij}$" se refieren a un puntero a la posición del elemento $\tilde{u}_{i,j}$ en la estructura de datos empleada para almacenar $\tilde{U}$.

para obtener un precondicionador eficaz es, en general, significativamente menor. De hecho, las listas enlazadas se emplean actualmente en paquetes de software punteros de precondicionado como ILUPACK.

Las reglas de descarte de la factorización ILUT(τ, p) (ver Algoritmo 2.6) se pueden incorporar de forma trivial a la factorización ILUC en las líneas 16 y 17 del Algoritmo 2.7. En este punto es necesario recalcar que las factorizaciones ILUT y ILUC *no obtienen generalmente la misma aproximación de* A, incluso cuando se emplean los mismos parámetros para las reglas de descarte. Por lo tanto, en el caso de las factorizaciones ILU con descarte basadas en la magnitud de los elementos, la variante algorítmica también juega un cierto papel en la calidad de la aproximación.

Factorizaciones de tipo ILDU y AINV

En este apartado presentamos una pareja de factorizaciones en el marco de las técnicas de precondicionado basadas en las factorizaciones aproximadas de A y A^{-1}. Para derivar ambas factorizaciones se utiliza una nueva terminología que resultará útil para el resto del apartado y de la tesis.

En primer lugar, consideramos la factorización LDU de A y la factorización incompleta ILDU (*Incomplete LDU*). El proceso de factorización LDU descompone la matriz de coeficientes del sistema en el producto $A = LDU$, donde $L \in \mathbb{R}^{n,n}$ es triangular inferior unidad, $D \in \mathbb{R}^{n,n}$ es diagonal, y $U \in \mathbb{R}^{n,n}$ es triangular superior unidad. Cuando A es simétrica se utiliza el término "factorización LDL^T" ya que $U = L^T$. Además, si es definida positiva, entonces todos los elementos de la diagonal principal de D son positivos. Consideremos la factorización $A = \hat{L}\hat{U}$, con $\text{diag}(\hat{U}) = (\hat{u}_{11}\hat{u}_{22}\dots\hat{u}_{nn})$, entonces $L = \hat{L}$, $DU = \hat{U}$ y, por lo tanto,

$$D = \begin{pmatrix} \hat{u}_{11} & 0 & 0 & \dots & 0 \\ 0 & \hat{u}_{22} & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & \hat{u}_{nn} \end{pmatrix}. \tag{2.14}$$

Explotando esta relación entre los factores L, D y U y los factores $\hat{L}$ y $\hat{U}$, la derivación de la variante algorítmica KIJ de la factorización LDU es inmediata a partir del Algoritmo 2.1. En concreto, solo es necesario añadir la sentencia $d_{kk} \leftarrow a_{kk}^{(k-1)}$ antes de la línea 3, cambiar la de la línea 4 por $u_{kj} \leftarrow a_{kj}^{(k-1)}/d_{kk}$ y la actualización de la línea 7 por $a_{ij}^{(k)} \leftarrow a_{ij}^{(k-1)} - l_{ik}d_{kk}u_{kj}$. Para derivar la variante de Crout y la IKJ de la factorización ILDU se efectúan cambios similares en los algoritmos correspondientes. Es fácil comprobar que, si particionamos la matriz de coeficientes como

$$A = \begin{pmatrix} \beta & d^T \\ c & E \end{pmatrix}, \tag{2.15}$$

con $\beta \in \mathbb{R}$, y c, d^T y E de tamaño acorde al de β, entonces se cumple que

$$A = \begin{pmatrix} \beta & d^T \\ c & E \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ p & I \end{pmatrix} \begin{pmatrix} \delta & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} 1 & q^T \\ 0 & I \end{pmatrix}, \tag{2.16}$$

con $\delta = d_{11}$, $p = \begin{pmatrix} l_{21} & \dots & l_{n1} \end{pmatrix}^T$, $q^T = \begin{pmatrix} u_{12} & \dots & u_{1n} \end{pmatrix}$ y $S = (a_{ij}^{(1)})_{i,j=2\dots n}$ construidos a partir de los elementos obtenidos tras ejecutar la primera iteración de la variante KIJ de la factorización LDU. La matriz $S = E - p\delta q^T$ se denomina *complemento de Schur* de β en A y corresponde a la matriz resultante de aplicar sobre A las actualizaciones correspondientes a la eliminación de c, es decir, los elementos de la primera columna de A situados en la parte triangular inferior. La ecuación (2.16) se denomina factorización LDU *parcial* ya que faltan por eliminar las $n-1$ columnas restantes. Esta factorización se completa aplicando recursivamente (2.15) y (2.16) a S.

La factorización parcial (2.16) se convierte en una factorización parcial incompleta ILDU descartando elementos de p y de q^T, típicamente aquellos de magnitud inferior a un umbral τ. Si denotamos como $\tilde{p}$ y $\tilde{q}^T$ al resultado de descartar algunos elementos de p y de q^T, respectivamente, entonces el complemento de Schur *aproximado* se define como

$$\tilde{S} = E - \tilde{p}\delta\tilde{q}^T \approx S. \tag{2.17}$$

La factorización parcial ILDU se completa aplicando recursivamente el mismo proceso a $\tilde{S}$. La variante KIJ de la factorización ILDU aparece en el Algoritmo 2.8. En la implementación eficiente del algoritmo, la actualización de rango 1 de la línea 12 se restringe a aquellos elementos $\tilde{s}_{ij}$ para los cuales $\tilde{p}_i \neq 0$ y $\tilde{q}_j \neq 0$. Esta actualización equivale a (2.17), ya que los primeros k elementos de $\tilde{p}$ y de $\tilde{q}$ son nulos y, por tanto, en la iteración k-ésima sólo se actualiza el bloque formado por la intersección de las últimas $n-k$ filas y columnas de la matriz $\tilde{S}$. Los vectores $\hat{p}$ y $\hat{q}$ son $\tilde{p}$ y $\tilde{q}$ antes de aplicar descarte, y aunque no se emplean realmente en los cálculos que efectúa el algoritmo, nos van a servir para el desarrollo venidero. A partir de ahora utilizamos el término factorización ILU para referirnos también a aproximaciones del tipo $A \approx \tilde{L}\tilde{D}\tilde{U}$.

Algoritmo 2.8: Variante KIJ de la factorización ILDU(τ)

1 $\tilde{L} \leftarrow I$, $\tilde{U} \leftarrow I$, $\tilde{S} \leftarrow A$
2 **para** $k = 1 : n$ **hacer**
3 $\quad \delta \leftarrow \tilde{s}_{kk}$
4 $\quad \hat{p}_i \leftarrow \tilde{p}_i \leftarrow 0$ **y** $\hat{q}_i \leftarrow \tilde{q}_i \leftarrow 0$ **con** $i = 1 : k$
5 $\quad \hat{p}_i \leftarrow \tilde{p}_i \leftarrow \tilde{s}_{ik}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
6 $\quad \hat{q}_i \leftarrow \tilde{q}_i \leftarrow \tilde{s}_{ki}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
7 $\quad \tilde{p}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{p}_i| \leq \tau$
8 $\quad \tilde{q}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{q}_i| \leq \tau$
9 $\quad \tilde{l}_{ik} \leftarrow \tilde{p}_i$ **con** $i = k+1 : n$, $\tilde{p}_i \neq 0$
10 $\quad \tilde{u}_{ki} \leftarrow \tilde{q}_i$ **con** $i = k+1 : n$, $\tilde{q}_i \neq 0$
11 $\quad \tilde{d}_{kk} \leftarrow \delta$
12 $\quad \tilde{S} \leftarrow \tilde{S} - \tilde{p}\delta\tilde{q}^T$
13 **fin para**

Las factorizaciones de tipo AINV [26, 28] (*Approximate INVerse*) obtienen los factores incompletos de A^{-1} *directamente*. Cuando no se aplica descarte de elementos durante el proceso, estas factorizaciones calculan dos matrices triangulares superiores unidad, $W \in \mathbb{R}^{n,n}$ y $Z \in \mathbb{R}^{n,n}$, y una matriz diagonal, $D \in \mathbb{R}^{n,n}$, de tal forma que $W^T AZ = D$; es decir, $A^{-1} = ZD^{-1}W^T$. Si A admite una factorización LDU, entonces se cumple que $W = L^{-T}$ y $Z = U^{-1}$. Por lo tanto,

$$W^T A = DU \quad \text{y} \quad Z^T A^T = DL^T. \tag{2.18}$$

Las relaciones entre las estructuras de las matrices en (2.18) se emplean para derivar la variante *right-looking* de la factorización AINV. Teniendo en cuenta que $W^T A$ ($Z^T A^T$) es triangular superior, las columnas $k+1, \ldots, n$ de W (Z) deben ser ortogonales a la columna k-ésima de A (A^T) para $k = 1, 2, \ldots, n$. La variante *right-looking* efectúa un proceso de ortogonalización de las columnas de W y Z en el que la estructura de $W^T A$ y $Z^T A^T$ se transforma en triangular superior paso a paso. Cuando en el proceso de ortogonalización se aplica descarte, entonces se obtienen dos factores triangulares superiores unidad, $\tilde{W} \in \mathbb{R}^{n,n}$ y $\tilde{Z} \in \mathbb{R}^{n,n}$, y una matriz diagonal, $\tilde{D} \in \mathbb{R}^{n,n}$, de tal forma que $\tilde{W}^T A\tilde{Z} \approx \tilde{D}$. Posteriormente, la matriz $M^{-1} = \tilde{Z}\tilde{D}^{-1}\tilde{W}^T$ se puede aplicar de forma *explícita* como precondicionador del sistema lineal (1.1), es decir, sin la necesidad de invertir los factores incompletos $\tilde{Z}$ y $\tilde{W}$, aumentando de esa forma la robustez numérica del método.

La variante *right-looking* de la factorización AINV se muestra en el Algoritmo 2.9. Al comienzo de la iteración k-ésima ya se han calculado las primeras k columnas de $\tilde{W}$ y de $\tilde{Z}$, y se procede con la actualización de las columnas $k+1, \ldots, n$. En la iteración i-ésima del bucle de la línea 6, la columna i-ésima de $\tilde{W}$ ($\tilde{Z}$) se actualiza con una combinación lineal de la columna k-ésima de $\tilde{W}$ ($\tilde{Z}$). El coeficiente $\tilde{p}_i$ ($\tilde{q}_i$) de esta combinación lineal se obtiene en la línea 7 de tal forma que la columna i-ésima de $\tilde{W}$ ($\tilde{Z}$) sea ortogonal a la columna k-ésima de A (A^T) después de la actualización efectuada por el bucle de la línea 8 (11). Tras ejecutarse el bucle de la línea 6, en la línea 15 (16) se aplica descarte en base a la magnitud de aquellos elementos de $\tilde{W}$ ($\tilde{Z}$) involucrados en las actualizaciones de la iteración k-ésima, es decir, de aquellos elementos que pertenecen al bloque formado por la intersección de las primeras i filas y las últimas $n-k$ columnas. En concreto, un elemento de este bloque de $\tilde{W}$ ($\tilde{Z}$) se descarta si su magnitud $|\tilde{w}_{ji}|$ ($|\tilde{z}_{ji}|$) es inferior a la de un umbral preestablecido τ. Como resultado de descartar estos elementos, las columnas $k+1, \ldots, n$ de $\tilde{W}$ ($\tilde{Z}$) son sólo *aproximadamente* ortogonales a la columna k-ésima de A (A^T).

Algoritmo 2.9: Variante *right-looking* de la factorización AINV(τ)

1 $\tilde{W} \leftarrow I$
2 $\tilde{Z} \leftarrow I$
3 **para** $k = 1 : n$ **hacer**
4 $\tilde{p}_j \leftarrow 0$ **y** $\tilde{q}_j \leftarrow 0$ **con** $j = 1 : k-1$
5 $\tilde{p}_k = e_k^T \tilde{W}^T A e_k$ **y** $\tilde{q}_k = e_k^T A \tilde{Z} e_k$
6 **para** $i = k+1 : n$ **hacer**
7 $\tilde{p}_i = e_i^T \tilde{W}^T A e_k / \tilde{p}_k$ **y** $\tilde{q}_i = e_i^T A \tilde{Z} e_k / \tilde{q}_j$
8 **para** $j = 1 : i, \tilde{w}_{jk} \neq 0$ **hacer**
9 $\tilde{w}_{ji} \leftarrow \tilde{w}_{ji} - \tilde{p}_i \tilde{w}_{jk}$
10 **fin para**
11 **para** $j = 1 : i, \tilde{z}_{jk} \neq 0$ **hacer**
12 $\tilde{z}_{ji} \leftarrow \tilde{z}_{ji} - \tilde{q}_i \tilde{z}_{jk}$
13 **fin para**
14 **fin para**
15 $\tilde{w}_{ji} \leftarrow 0$ **con** $i = k+1 : n,\ j = 1 : i,\ |\tilde{w}_{ji}| \leq \tau$
16 $\tilde{z}_{ji} \leftarrow 0$ **con** $i = k+1 : n,\ j = 1 : i,\ |\tilde{z}_{ji}| \leq \tau$
17 $\tilde{d}_{kk} \leftarrow p_k$
18 **fin para**

Descarte en base a la magnitud de la inversa de los factores: factorización ILDU_INV

Las reglas de descarte basadas en la magnitud de los elementos de los factores representan un buen compromiso para muchas aplicaciones numéricas. Sin embargo, existen ciertos problemas, como por ejemplo aquellos en los que A es altamente indefinida, que son extremadamente sensibles al valor de τ y en los que únicamente es posible obtener precondicionadores eficaces seleccionando valores de τ muy pequeños. Como consecuencia, la factorización ILU puede sufrir mucho llenado y, por tanto, puede tener un coste computacional inaceptable. En [29, 30, 33] se desarrollaron reglas de descarte *nuevas* para este tipo de problemas que tienen por objeto obtener precondicionadores ILU menos sensibles a los errores que se producen al descartar elementos de los factores. Estas reglas de descarte se conocen en la literatura como *inverse-based dropping rules*. En el presente documento utilizaremos la traducción literal al lenguaje castellano de este término, es decir, *reglas de descarte basadas en la inversa*. En [29, 30] estas reglas se evaluaron en combinación con técnicas de pivotamiento prestadas del área de los métodos directos. En [110] las reglas de descarte basadas en la inversa se implementaron eficientemente en el marco de la

factorización ILUC, aunque en este trabajo no se empleó pivotamiento. Por simplicidad, en este apartado nos limitamos a presentar las reglas de descarte y obviamos el uso de pivotamiento en los métodos numéricos. Es posible encontrar más detalles en el Apartado 2.1.2.

Para comprender la causa por la cual las reglas de descarte empleadas por el Algoritmo 2.8 no son tan fiables como las empleadas por el Algoritmo 2.9, es necesario inspeccionar la forma en la que las factorizaciones ILU se emplean para precondicionar el sistema (1.1). Si $\tilde{L}$, $\tilde{D}$, y $\tilde{U}$ son los factores incompletos calculados por el Algoritmo 2.8, y R la matriz residuo de la factorización, entonces

$$\tilde{L}^{-1}A\tilde{U}^{-1} = \tilde{D} + \tilde{L}^{-1}R\tilde{U}^{-1}. \tag{2.19}$$

En esta expresión, $\tilde{L}^{-1}A\tilde{U}^{-1}$ es la matriz del sistema precondicionado, es decir, la matriz del sistema que resuelve el método iterativo. Desde el punto de vista de la convergencia de los métodos iterativos basados en la generación de subespacios de Krylov, interesa que $\tilde{L}^{-1}A\tilde{U}^{-1} \approx \tilde{D}$. El problema de las reglas de descarte basadas en la magnitud de los elementos es que no tienen en cuenta, y por tanto no controlan, el efecto que tiene descartar elementos de $\tilde{L}$ y de $\tilde{U}$ sobre la magnitud de $\tilde{L}^{-1}R\tilde{U}^{-1}$, ya que $\tilde{L}^{-1}$ y $\tilde{U}^{-1}$ no están disponibles durante el proceso. Minimizar *vorazmente* $\|R\|$, descartando los elementos de menor magnitud, puede ocasionar que $\|\tilde{L}^{-1}R\tilde{U}^{-1}\|$ crezca de forma arbitraria [32], obteniendo en ese caso una factorización ILU ineficaz como precondicionador del sistema. Sin embargo, en el caso del Algoritmo 2.9, al descartar elementos de los factores $\tilde{W}$ y $\tilde{Z}$ de magnitud inferior a τ, tenemos control directo sobre la magnitud del error $\tilde{R} \in \mathbb{R}^{n,n}$ causado en la matriz del sistema precondicionado $\tilde{W}^T A\tilde{Z} = \tilde{D}+\tilde{R}$ al descartar elementos en el proceso de aproximación. La experimentación revela que, especialmente cuando A es altamente indefinida, los precondicionadores derivados de factorizaciones del tipo AINV son más robustos que los derivados de factorizaciones ILU con descarte en base a la magnitud de los elementos (consultar, por ejemplo [32]).

Las reglas de descarte basadas en la inversa se derivan a partir de las relaciones entre las factorizaciones ILU y las factorizaciones AINV. En [33] se demostró que, transformando el Algoritmo 2.8, es posible calcular dos factores incompletos $\tilde{L}$ y $\tilde{U}$ de tal forma que $\tilde{L}^{-T} \approx \tilde{W}$ y $\tilde{U}^{-1} \approx \tilde{Z}$, con $\tilde{W}$ y $\tilde{Z}$ los factores calculados por el Algoritmo 2.9. A continuación describimos en qué consiste esta transformación, y cómo se puede explotar para derivar las reglas de descarte basadas en la inversa. Para más detalles, incluyendo la justificación correspondiente, referirse a [33].

Consideremos en primer lugar cómo se puede derivar un proceso para invertir $\tilde{L}$ y $\tilde{U}$ progresivamente en el Algoritmo 2.8. Si $\tilde{U}_{k-1}$ se refiere al contenido del factor $\tilde{U}$ después de las primeras $k-1$ iteraciones, entonces es fácil comprobar que tiene la siguiente estructura

$$\tilde{U}_{k-1} = \left(\begin{array}{ccc|c|ccc} 1 & \tilde{u}_{1,2} & \dots & \tilde{u}_{1,k} & \dots & \dots & \tilde{u}_{1,n} \\ & \ddots & \ddots & \vdots & & & \vdots \\ & & 1 & \tilde{u}_{k-1,k} & \dots & \dots & \tilde{u}_{k-1,n} \\ \hline & & & 1 & 0 & \dots & 0 \\ \hline & & & & 1 & \ddots & \vdots \\ & & & & & \ddots & 0 \\ & & & & & & 1 \end{array}\right) \equiv \left(\begin{array}{c|c|c} A & b & C \\ \hline 0 & v & w^T \\ \hline 0 & 0 & Z \end{array}\right). \tag{2.20}$$

En la iteración k-ésima, el Algoritmo 2.8 calcula los elementos situados en la posición del vector w^T en el particionado anterior y, posteriormente, añade estos elementos a $\tilde{U}_{k-1}$ para obtener $\tilde{U}_k$. Si $\tilde{q} = \begin{pmatrix} 0 & \dots & 0 & \tilde{q}_{k+1} & \dots & \tilde{q}_n \end{pmatrix}^T$ es el vector calculado en la iteración k-ésima del Algoritmo 2.8, entonces

$$\tilde{U}_k = \tilde{U}_{k-1} + e_k\tilde{q}^T. \tag{2.21}$$

Debido a la estructura de $\tilde{U}_{k-1}$ y de $\tilde{q}$ se cumple que $\tilde{q}^T\tilde{U}_{k-1} = \tilde{q}^T$ y, por lo tanto,

$$\tilde{U}_k = (I + e_k\tilde{q}^T)\tilde{U}_{k-1}. \tag{2.22}$$

Invirtiendo (2.22) obtenemos que

$$\tilde{U}_k^{-1} = \tilde{U}_{k-1}^{-1}(I + e_k\tilde{q}^T)^{-1} = \tilde{U}_{k-1}^{-1}(I - e_k\tilde{q}^T). \tag{2.23}$$

Procediendo de forma homóloga para el factor $\tilde{L}$, entonces

$$\tilde{L}_k^{-T} = \tilde{L}_{k-1}^{-T}(I - e_k\tilde{p}^T), \tag{2.24}$$

con $\tilde{p} = \begin{pmatrix} 0 & \dots & 0 & \tilde{p}_{k+1} & \dots & \tilde{p}_n \end{pmatrix}^T$ el vector calculado en la iteración k-ésima del Algoritmo 2.8. La recurrencia derivada a partir de (2.23) ((2.24)), comenzando por $\tilde{U}_0 = \tilde{U}_0^{-1} = I$ ($\tilde{L}_0^T = \tilde{L}_0^{-T} = I$), para $k = 1, 2, \dots, n-1$, permite invertir progresivamente el factor $\tilde{U}$ ($\tilde{L}^T$) durante el proceso de factorización ILU.

En general, no existen garantías de que $\tilde{U}^{-1} \approx \tilde{Z}$ ($\tilde{L}^{-T} \approx \tilde{W}$) a no ser que, para empezar, introduzcamos ciertos cambios en esta recurrencia [33]. El primer cambio considera para (2.23) ((2.24)) el vector $\hat{q}$ ($\hat{p}$) obtenido en la iteración k-ésima del Algoritmo 2.8, en lugar del vector $\tilde{q}$ ($\tilde{p}$). Si las matrices $\hat{Z}_0, \hat{Z}_1, \dots, \hat{Z}_{n-1}$ ($\hat{W}_0, \hat{W}_1, \dots, \hat{W}_{n-1}$), con $\hat{Z} = \hat{Z}_{n-1}$ ($\hat{W} = \hat{W}_{n-1}$), se refieren a las matrices de la recurrencia (2.23) ((2.24)) después de este cambio, entonces

$$\hat{Z}_k = (\hat{z}_{ij}^{(k)}) = \hat{Z}_{k-1}(I - e_k\hat{q}^T) \qquad (\hat{W}_k = (\hat{w}_{ij}^{(k)}) = \hat{W}_{k-1}(I - e_k\hat{p}^T)). \tag{2.25}$$

El segundo cambio consiste en descartar aquellos elementos $\hat{z}_{ij}^{(k)}$ ($\hat{w}_{ij}^{(k)}$) para los cuales $1 \le i \le k$ y $k+1 \le j \le n$, y $|\hat{z}_{ij}^{(k)}| \le \tau$ ($|\hat{w}_{ij}^{(k)}| \le \tau$). Es fácil comprobar que estos elementos pertenecen al bloque de $\hat{Z}_{k-1}$ ($\hat{W}_{k-1}$) que (2.25) actualiza para obtener $\hat{Z}_k$ ($\hat{W}_k$). Por simplicidad en la notación, empleamos también $\hat{Z}_k$ ($\hat{W}_k$) para referirnos a la matriz resultante al descartar estos elementos.

El teorema central de [33] enuncia que, *cambiando las reglas de descarte para los factores* $\tilde{L}$ *y* $\tilde{U}$ en el Algoritmo 2.8, se puede conseguir que $\hat{Z} \approx \tilde{L}^{-T}$ y $\hat{W} \approx \tilde{U}^{-1}$ en "algún sentido". En concreto, si $\tilde{l}_{ik}$ se descarta, para $k = 1, \dots, n$, $i > k$, cuando

$$|\tilde{l}_{ik}| \cdot \|\hat{W}_k e_k\| \le \tau, \tag{2.26}$$

y $\tilde{u}_{ki}$ cuando

$$|\tilde{u}_{ki}| \cdot \|\hat{Z}_k e_k\| \le \tau, \tag{2.27}$$

se cumple que $\hat{Z} \approx \tilde{L}^{-T}$ y $\hat{W} \approx \tilde{U}^{-1}$ en el siguiente sentido

$$|(I - \hat{Z}\tilde{L}^T)_{rs}| \le 2(r-s)\tau, \ |(I - \hat{W}\tilde{U})_{rs}| \le 2(r-s)\tau, \tag{2.28}$$

para $r = 1, \dots, n$ y $s = r+1, \dots, n$. $\hat{Z}\tilde{L}^T$ y $\hat{W}\tilde{U}$ son matrices triangulares superiores unidad y, por lo tanto, $(I - \hat{Z}\tilde{L}^T)_{rs} = 0$ para $r = 1, \dots, n$ y $s = 1, \dots, r$. Además, el teorema también enuncia que, definiendo el complemento de Schur aproximado en la iteración k-ésima del Algoritmo 2.8 como el bloque formado por la intersección de las $n-k$ filas y columnas de la matriz,

$$\hat{W}_k^T A \hat{Z}_k, \tag{2.29}$$

entonces $\hat{W} = \tilde{W}$ y $\hat{Z} = \tilde{W}$, es decir, los factores $\hat{W}$ y $\hat{Z}$ calculados aproximando progresivamente $\tilde{L}^{-T}$ y $\tilde{U}^{-1}$ en el Algoritmo 2.8 *equivalen a los factores calculados por el Algoritmo 2.9*.

Las consideraciones de los párrafos anteriores se plasman en el Algoritmo 2.10, que incorpora en las líneas 7 y 10 el proceso basado en la recurrencia (2.25). Además, los elementos de $\tilde{L}$ y $\tilde{U}$ se descartan en las líneas 12 y 13 conforme a (2.26) y (2.27), respectivamente, y el complemento de Schur aproximado se calcula según (2.29) en la línea 17. Las inversas de los factores incompletos $\tilde{L}^T$ y $\tilde{U}$ calculados por el Algoritmo 2.10 son aproximaciones en el sentido (2.28) de los factores $\tilde{W}$ y $\tilde{Z}$ calculados por el Algoritmo 2.9.

Algoritmo 2.10: Variante KIJ de la factorización ILDU(τ)-AINV(τ).

$\tilde{L} \leftarrow I$, $\tilde{U} \leftarrow I$, $\tilde{S} \leftarrow A$
$\hat{W}_0 \leftarrow I$, $\hat{Z}_0 \leftarrow I$, $\tilde{S} \leftarrow A$
para $k = 1 : n$ **hacer**
  $\delta \leftarrow \tilde{s}_{kk}$
  $\hat{p}_i \leftarrow \tilde{p}_i \leftarrow 0$ **y** $\hat{q}_i \leftarrow \tilde{q}_i \leftarrow 0$ **con** $i = 1 : k$
  $\hat{p}_i \leftarrow \tilde{p}_i \leftarrow \tilde{s}_{ik}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
  $\hat{W}_k = \hat{W}_{k-1}(I - e_k \hat{p}^T)$
  $\tilde{w}_{ij}^{(k)} \leftarrow 0$ **con** $i = 1 : k,\ j = k+1 : n,\ |\tilde{w}_{ij}^{(k)}| \leq \tau$
  $\hat{q}_i \leftarrow \tilde{q}_i \leftarrow \tilde{s}_{ki}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
  $\hat{Z}_k = \hat{Z}_{k-1}(I - e_k \hat{q}^T)$
  $\tilde{z}_{ij}^{(k)} \leftarrow 0$ **con** $i = 1 : k,\ j = k+1 : n,\ |\tilde{z}_{ij}^{(k)}| \leq \tau$
  $\tilde{p}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{p}_i| \cdot \|\hat{W} e_k\| \leq \tau$
  $\tilde{q}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{q}_i| \cdot \|\hat{Z} e_k\| \leq \tau$
  $\tilde{l}_{ik} \leftarrow \tilde{p}_i$ **con** $i = k+1 : n$, $\tilde{p}_i \neq 0$
  $\tilde{u}_{ki} \leftarrow \tilde{q}_i$ **con** $i = k+1 : n$, $\tilde{q}_i \neq 0$
  $\tilde{d}_{kk} \leftarrow \delta$
  $\tilde{s}_{ij} \leftarrow (\hat{W}_k^T A \hat{Z}_k)_{k+i,k+j}$ **con** $i, j = 1 : n-k$
fin para

La proximidad de $\hat{W}$ a $\tilde{L}^{-T}$ y de $\hat{Z}$ a $\tilde{U}^{-1}$ en el sentido (2.28) sugiere que, en las reglas de descarte (2.26) y (2.27), $\|e_k^T \tilde{L}^{-1}\|$ y $\|\tilde{U}^{-1} e_k\|$ pueden ejercer como de $\|\hat{W} e_k\|$ y $\|\hat{Z} e_k\|$, respectivamente. Tras aplicar esta substitución, y tomando las normas infinito $\|e_k^T \tilde{L}^{-1}\|$ y $\|\tilde{U}^{-1} e_k\|$, entonces las *reglas de descarte basadas en la inversa* descartan un elemento $\tilde{l}_{ik}$, con $i > k$ cuando

$$|\tilde{l}_{ik}| \cdot \|e_k^T \tilde{L}^{-1}\|_\infty \leq \tau, \tag{2.30}$$

y un elemento $\tilde{u}_{ki}$ cuando

$$|\tilde{u}_{ki}| \cdot \|\tilde{U}^{-1} e_k\|_\infty \leq \tau. \tag{2.31}$$

De esta forma se puede evitar el elevado coste computacional y de almacenamiento requerido para calcular $\hat{W}$ y $\hat{Z}$ de forma explícita en el Algoritmo 2.10.

Para descartar aquellos elementos $\tilde{l}_{ik}$ ($\tilde{u}_{ki}$) que cumplan (2.30) ((2.31)) en la iteración k-ésima de la factorización ILU, necesitamos estimaciones de la norma infinito de la fila (columna) k-ésima de $\tilde{L}^{-1}$ ($\tilde{U}^{-1}$), es decir, estimaciones de $\|e_k^T \tilde{L}^{-1}\|_\infty$ ($\|\tilde{U}^{-1} e_k\|_\infty$), ya que la inversa de $\tilde{L}$ ($\tilde{U}$) no está disponible durante el proceso. En principio, este problema requiere estimar la norma de las filas de la inversa de una matriz triangular inferior L (superior U), independientemente de si L proviene o no de una factorización ILU. En [53, 76] se desarrollaron algoritmos generales para estimar el número de condición de matrices densas con estructura triangular superior, y en [29, 30] estos algoritmos se adaptaron para estimar la norma de las filas (columnas) de la inversa de una matriz dispersa con estructura triangular inferior (superior).

Supongamos que $L \in \mathbb{R}^{n,n}$ es una matriz dispersa con estructura triangular inferior unidad, y consideremos que deseamos obtener estimaciones de $\|e_k^T L^{-1}\|_\infty$, para $k = 1, \ldots, n$. Si tenemos en cuenta que la norma matricial infinito [76] de $e_k^T L^{-1} \in \mathbb{R}^{1,n}$ se define como

$$\|e_k^T L^{-1}\|_\infty = \max_{\|b\|_\infty = 1} \|e_k^T L^{-1} b\|_\infty = \max_{\|b\|_\infty = 1} |e_k^T L^{-1} b|, \tag{2.32}$$

entonces para todo vector $\hat{b}$, cual que $\|\hat{b}\|_\infty = 1$, se cumple que $\|e_k^T L^{-1}\|_\infty \geq |e_k^T L^{-1} \hat{b}|$. Si obtenemos un vector $\hat{b}$ para el que la magnitud de $|e_k^T L^{-1} \hat{b}|$ está cerca de su cota superior

$\|e_k^T L^{-1}\|_\infty$, con $k = 1, \ldots, n$, entonces las componentes de la solución al sistema $Lx = \hat{b}$ se pueden emplear como estimadores de $\|e_k^T L^{-1}\|_\infty$; es decir, $|x_k| \approx \|e_k^T L^{-1}\|_\infty$, para $k = 1, \ldots, n$.

En los algoritmos que se desarrollaron en [29, 30], las componentes del vector $\hat{b} = (b)_k$ toman los valores ± 1, y el signo de estas componentes se escoge dinámicamente en el proceso de substitución progresiva con el propósito de maximizar (heurísticamente) las componentes del vector $|x|$. Tras ejecutar las primeras $k-1$ iteraciones del proceso de substitución progresiva, se obtiene un vector $v^{(k-1)}$ que cumple que

$$Lv^{(k-1)} = L \begin{pmatrix} x_1 \\ \vdots \\ x_{k-1} \\ v_k^{(k-1)} \\ \vdots \\ v_n^{(k-1)} \end{pmatrix} = \begin{pmatrix} \hat{b}_1 \\ \vdots \\ \hat{b}_{k-1} \\ 0 \\ \vdots \\ 0 \end{pmatrix}. \tag{2.33}$$

En la iteración k-ésima, la componente k-ésima de x se obtiene como $x_k = \hat{b}_k - v_k^{(k-1)}$ y, posteriormente, las componentes $v_{k+1}^{(k)}, \ldots, v_n^{(k)}$ de $v^{(k)}$ se obtienen a partir de las del vector $v^{(k-1)}$ mediante la siguiente actualización

$$\begin{pmatrix} v_{k+1}^{(k)} \\ \vdots \\ v_n^{(k)} \end{pmatrix} \leftarrow \begin{pmatrix} v_{k+1}^{(k-1)} \\ \vdots \\ v_n^{(k-1)} \end{pmatrix} + \begin{pmatrix} l_{k+1,k} \\ \vdots \\ l_{n,k} \end{pmatrix} x_k. \tag{2.34}$$

Por lo tanto, teniendo en cuenta que $\hat{b}_k = 1$ o $\hat{b}_k = -1$, x_k puede tomar los valores $x_k^+ = 1 - v_k^{(k-1)}$ o $x_k^- = -1 - v_k^{(k-1)}$, respectivamente. Asimismo, las componentes $v_{k+1}^{(k)}, \ldots, v_n^{(k)}$ pueden tomar dos valores alternativos en función de si empleamos x_k^+ o x_k^- en (2.34).

Desde el punto de vista de la calidad del estimador de $\|e_k^T L^{-1}\|_\infty$ interesa maximizar $|x_k|$. Por ejemplo, si $|x_k^+| = \max(|x_k^+|, |x_k^-|)$, entonces conviene que $x_k = x_k^+$ en el paso k-ésimo del proceso de substitución. Sin embargo, desde el punto de vista de la calidad de las estimaciones de la norma del resto de las filas de L^{-1}, interesa aquel valor de x_k para el que la magnitud de las componentes $|v_{k+1}^{(k)}|, \ldots, |v_n^{(k)}|$ sea lo más grande posible. El algoritmo diseñado en [29] emplea un heurístico que escoge el signo de la componente $\hat{b}_k$ con el objetivo de maximizar la función $|x_k| + \|(v_{k+1}^{(k)}, \ldots, v_n^{(k)})^T\|_1$. El algoritmo desarrollado en [30] emplea una variante de este heurístico en la que este signo se escoge para maximizar exclusivamente $\|(v_{k+1}^{(k)}, \ldots, v_n^{(k)})^T\|_1$. Sin embargo, después de escoger x_k con este propósito y actualizar las componentes $(v_{k+1}^{(k)}, \ldots, v_n^{(k)})^T$ de forma acorde a la elección para x_k, se acaba usando $\max(|x_k^+|, |x_k^-|)$ como estimador de $\|e_k^T L^{-1}\|_\infty$. Esta segunda técnica se muestra en el Algoritmo 2.11. Este algoritmo produce como resultado estimadores $t_k^L \approx \|e_k^T L^{-1}\|_\infty$, para $k = 1, \ldots, n$. La estructura dispersa de la matriz L se explota en las líneas 6 y 7 calculando $\|(v_{k+1}^{(k)}, \ldots, v_n^{(k)})^T\|_1$ exclusivamente para aquellas componentes $v_j^{(k)}$, con $j > k$, para las cuales $j \in \mathcal{P}$. Es decir, para aquellas componentes para las cuales $l_{jk} \neq 0$ en (2.34). Además, en el bucle de la línea 14 sólo se actualizan aquellas componentes $v_j^{(k)}$ que cumplen esta propiedad. El coste computacional del Algoritmo 2.11 es del mismo orden que el coste requerido para resolver un sistema de la forma $Lr = s$ mediante el proceso de substitución progresiva.

El cálculo efectuado por el Algoritmo 2.11 se puede combinar con el Algoritmo 2.8 teniendo en cuenta que, en la iteración k-ésima de este último, se dispone de la información requerida para estimar $\|e_k^T \tilde{L}^{-1}\|_\infty$ y $\|\tilde{U}^{-1} e_k\|_\infty$, es decir, las primeras k columnas y filas de $\tilde{L}$ y $\tilde{U}$, respectivamente. La factorización ILDU_INV(τ) se muestra en el Algoritmo 2.12. Esta factorización

Algoritmo 2.11: Estimador de $\|e_k^T L^{-1}\|_\infty$, con $k = 1 : n$.

1 $v^{(0)} = (v_1^{(0)}, \ldots, v_n^{(0)})^T = (0, \ldots, 0)^T$
2 **para** $k = 1 : n$ **hacer**
3 $\quad x^+ \leftarrow 1 - v_k^{(k-1)}$
4 $\quad x^- \leftarrow -1 - v_k^{(k-1)}$
5 $\quad \mathcal{P} \leftarrow \{m : k+1 \leq m \leq n, l_{mk} \neq 0\}$
6 $\quad v^+ \leftarrow \|(v_j^{(k-1)} + l_{jk}x^+)_{j\in\mathcal{P}}\|_1$
7 $\quad v^- \leftarrow \|(v_j^{(k-1)} + l_{jk}x^-)_{j\in\mathcal{P}}\|_1$
8 $\quad$ **si** $v^+ > v^-$ **entonces**
9 $\quad\quad x_k \leftarrow x^+$
10 $\quad$ **si no**
11 $\quad\quad x_k \leftarrow x^-$
12 $\quad$ **fin si**
13 $\quad v_j^{(k)} = v_j^{(k-1)}$ **con** $j = k+1 : n, l_{jk} = 0$
14 $\quad$ **para** $j \in \mathcal{P}$ **hacer**
15 $\quad\quad v_j^{(k)} \leftarrow v_j^{(k-1)} + l_{jk}x_k$
16 $\quad$ **fin para**
17 $\quad t_k^L \leftarrow \text{máx}(|x^+|, |x^-|)$
18 **fin para**

incorpora el cálculo de los estimadores a la factorización ILU con el objetivo de descartar elementos en base a la magnitud de las inversas de los factores incompletos. En concreto, $\tilde{l}_{ik}$, con $i > k$, se descarta en la línea 21 cuando

$$|\tilde{l}_{ik}| \cdot t_k^L \leq \tau, \tag{2.35}$$

y $\tilde{u}_{ki}$ en la línea 38 cuando

$$|\tilde{u}_{ki}| \cdot t_k^U \leq \tau, \tag{2.36}$$

con t_k^L y t_k^U los estimadores para $\|e_k^T \tilde{L}^{-1}\|_\infty$ y $\|\tilde{U}^{-1} e_k\|_\infty$ en (2.30) y (2.31), respectivamente. Aunque el Algoritmo 2.12 está basado en la variante KIJ de la factorización ILU, el cálculo de estos estimadores también se puede combinar con la variante ILUC [110]. De hecho, el núcleo de factorización de la biblioteca ILUPACK [35] incorpora por defecto este cálculo a la variante ILUC. Sin embargo, cabe destacar que el método empleado en este núcleo efectúa, por cada iteración k del Algoritmo 2.11, una segunda ronda en la que se intenta mejorar la calidad del estimador escogiendo el signo de $\hat{b}_k$ de forma alternativa [30]. Además, la factorización ILU en ILUPACK incorpora pivotamiento para controlar la magnitud de $\|e_k^T \tilde{L}^{-1}\|_\infty$ y $\|\tilde{U}^{-1} e_k\|_\infty$ durante el proceso de factorización con el objetivo de mejorar la calidad del precondicionador. Para más detalles, consultar [30, 34].

2.1.2. Fiabilidad de las factorizaciones ILU: existencia y estabilidad

Eliminación Gaussiana

En aritmética exacta la eliminación Gaussiana se completa *sin divisiones por cero* si y solo si todos los menores principales de A son no singulares. En este caso se dice que A *admite* una factorización LU o, indistintamente, que la factorización LU de A *existe*. Si la matriz A no admite una factorización LU, pero no es singular, entonces *siempre* es posible encontrar una matriz de permutación $P \in \mathbb{R}^{n,n}$ de tal forma que la matriz PA admita una factorización LU. Además,

Algoritmo 2.12: Factorización ILDU_INV(τ)

$\tilde{L} \leftarrow I$, $\tilde{U} \leftarrow I$, $\tilde{S} \leftarrow A$
$v^{(0)} = (v_1^{(0)}, \ldots, v_n^{(0)})^T = (0, \ldots, 0)^T$
$w^{(0)} = (w_1^{(0)}, \ldots, w_n^{(0)})^T = (0, \ldots, 0)^T$
para $k = 1 : n$ **hacer**
    $\delta \leftarrow \tilde{s}_{kk}$
    $\tilde{q}_i \leftarrow 0$ **con** $i = 1 : k$ ▷ Generación de la columna k-ésima de $\tilde{L}$
    $\tilde{p}_i \leftarrow \tilde{s}_{ik}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
    $x^+ \leftarrow 1 - v_k^{(k-1)}$, $x^- \leftarrow -1 - v_k^{(k-1)}$
    $\mathcal{P}_L \leftarrow \{m : k+1 \leq m \leq n, \tilde{p}_m \neq 0\}$
    $v^+ \leftarrow \|(v_j^{(k-1)} + \tilde{l}_{jk}x^+)_{j\in\mathcal{P}_L}\|_1$, $v^- \leftarrow \|(v_j^{(k-1)} + \tilde{l}_{jk}x^-)_{j\in\mathcal{P}_L}\|_1$
    si $v^+ > v^-$ **entonces**
        $x_k \leftarrow x^+$
    si no
        $x_k \leftarrow x^-$
    fin si
    $v_j^{(k)} = v_j^{(k-1)}$ **con** $j = k+1 : n$, $\tilde{l}_{jk} = 0$
    para $j \in \mathcal{P}_L$ **hacer**
        $v_j^{(k)} \leftarrow v_j^{(k-1)} + \tilde{l}_{jk}x_k$
    fin para
    $t_k^L \leftarrow \max(|x^+|, |x^-|)$
    $\tilde{p}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{p}_i| \cdot t_k^L \leq \tau$
    $\tilde{l}_{ik} \leftarrow \tilde{p}_i$ **con** $i = k+1 : n$, $\tilde{p}_i \neq 0$
    $\tilde{q}_i \leftarrow 0$ **con** $i = 1 : k$ ▷ Generación de la fila k-ésima de $\tilde{U}$
    $\tilde{q}_i \leftarrow \tilde{s}_{ki}/\tilde{s}_{kk}$ **con** $i = k+1 : n$
    $y^+ \leftarrow 1 - w_k^{(k-1)}$, $y^- \leftarrow -1 - w_k^{(k-1)}$
    $\mathcal{P}_U \leftarrow \{m : k+1 \leq m \leq n, \tilde{q}_m \neq 0\}$
    $w^+ \leftarrow \|(w_j^{(k-1)} + \tilde{u}_{kj}y^+)_{j\in\mathcal{P}_U}\|_1$, $w^- \leftarrow \|(w_j^{(k-1)} + \tilde{u}_{kj}y^-)_{j\in\mathcal{P}_U}\|_1$
    si $w^+ > w^-$ **entonces**
        $y_k \leftarrow y^+$
    si no
        $y_k \leftarrow y^-$
    fin si
    $w_j^{(k)} = w_j^{(k-1)}$ **con** $j = k+1 : n$, $\tilde{u}_{kj} = 0$
    para $j \in \mathcal{P}_U$ **hacer**
        $w_j^{(k)} \leftarrow w_j^{(k-1)} + \tilde{u}_{kj}y_k$
    fin para
    $t_k^U \leftarrow \max(|y^+|, |y^-|)$
    $\tilde{q}_i \leftarrow 0$ **con** $i = k+1 : n$, $|\tilde{q}_i| \cdot t_k^U \leq \tau$
    $\tilde{u}_{ki} \leftarrow \tilde{q}_i$ **con** $i = k+1 : n$, $\tilde{q}_i \neq 0$
    $\tilde{d}_{kk} \leftarrow \delta$
    $\tilde{S} \leftarrow \tilde{S} - \tilde{p}\delta\tilde{q}^T$
fin para

cabe destacar que si A es SDP, entonces A admite incondicionalmente una factorización de Cholesky. En [76, 164] es posible encontrar una descripción más detallada de estas propiedades.

En aritmética finita, el análisis numérico recurre *el condicionamiento del sistema* y *la estabilidad regresiva del resolutor* para estudiar el efecto de los errores de redondeo [94]. El condicionamiento del sistema refleja su sensibilidad a las perturbaciones en su entrada, es decir, a perturbaciones en A y b, y se cuantifica a través del número de condición [76, 164] de A, $\kappa(A)$. Cuando $\kappa(A)$ es un número "pequeño" (en relación a la precisión de la máquina), se dice que el sistema está *bien condicionado*. Los sistemas que cumplen esta propiedad son poco sensibles a cambios en su entrada, de tal forma que pequeñas perturbaciones de A y de b causan sólo pequeñas perturbaciones en la solución del sistema. Si, por el contrario, $\kappa(A)$ es un número "grande", se dice que el sistema está mal condicionado, y en este caso puede ocurrir que pequeñas perturbaciones en la entrada originen perturbaciones de magnitud mucho más grande en la solución del sistema. El estudio de la sensibilidad del sistema es útil si tenemos en cuenta que, en aritmética finita, obtenemos la solución $\hat{x}$ a un sistema perturbado $(A+\delta A)\hat{x} = b$, donde δA es una matriz de perturbación causada por los errores de redondeo que se producen en la factorización de A y la posterior resolución de los sistemas triangulares. Los errores de redondeo son inevitables en un computador y, por lo tanto, no podemos esperar resolver el sistema de forma precisa si el sistema está mal condicionado. Por esta razón, *los sistemas mal condicionados son problemas difíciles de resolver en un computador*. Así pues, el objetivo en álgebra lineal numérica es desarrollar métodos *regresivamente estables*, es decir, métodos en los que la magnitud de la perturbación sea *relativamente* "pequeña" con respecto a la de A. De forma más precisa, la magnitud de δA se considera "pequeña" si el cociente $\|\delta A\|/\|A\|$ es del orden de un múltiplo modesto de un número conocido como error de redondeo unidad, al que denotamos con la letra u. Cabe destacar que u es un número muy pequeño en los sistemas de representación de números reales en punto flotante de los computadores actuales. (Por ejemplo, $u = 2^{-53} \approx 1,1102 \times 10^{-16}$ en el sistema IEEE 754 de precisión doble.)

La magnitud de δA depende de los errores de redondeo cometidos en la factorización de A y durante la posterior resolución de los sistemas triangulares. Esta segunda computación no es crítica, ya que existen métodos sencillos y eficientes (como por ejemplo, las substituciones regresiva y progresiva) que son regresivamente estables. Sin embargo, el proceso de factorización es significativamente más crítico. De hecho, el proceso de eliminación Gaussiana no es regresivamente estable. La teoría existente establece que la estabilidad de la eliminación Gaussiana depende de la magnitud de las entradas de los factores L y U. En concreto, si el producto $\|L\|_\infty\|U\|_\infty$ es un múltiplo modesto de $\|A\|_\infty$, entonces se puede demostrar que el cociente $\|\delta A\|_\infty/\|A\|_\infty$ es un múltiplo modesto de u. Sin embargo, $\|L\|_\infty\|U\|_\infty/\|A\|_\infty$ puede ser arbitrariamente grande, dado que, entre otros factores, el método básico no emplea ninguna técnica para evitar elementos pivote $a_{kk}^{(k)}$ muy pequeños (en relación a los elementos de la columna k-ésima de A_k que están por debajo de la diagonal principal). El uso de elementos pivote muy pequeños puede resultar en multiplicadores muy grandes, que son a su vez las entradas de L. Por lo tanto, $\|L\|_\infty$ puede ser arbitrariamente grande. Estos multiplicadores muy grandes causan que múltiplos muy grandes de la fila k-ésima de A_k se sumen a las filas $k+1, \ldots, n$ y, como consecuencia, $\|U\|_\infty$ también puede ser arbitrariamente grande. Aunque en este caso no se puede emplear la teoría para derivar conclusión alguna sobre la estabilidad regresiva del proceso, la experiencia revela que, cuando en el proceso de eliminación surgen elementos pivote muy pequeños, existe una mayor probabilidad de que los errores de redondeo reduzcan significativamente la precisión del resultado. Para matrices SDP existe un resultado teórico similar para la factorización de Cholesky $A = LL^T$. En este caso la teoría asegura que la factorización de Cholesky es un proceso regresivamente estable, ya que $\|L\|_\infty\|L^T\|_\infty/\|A\|_\infty$ es del orden de un múltiplo modesto del error de redondeo unidad [164].

Para mejorar la estabilidad regresiva del proceso de eliminación Gaussiana de matrices no simétricas y/o indefinidas, es necesario incorporar técnicas de *pivotamiento*. Estas técnicas tienen

por objeto mantener acotada la magnitud de los elementos de los factores. Una de las técnicas de pivotamiento más usadas en la práctica es el pivotamiento parcial. En la iteración k-ésima del proceso de eliminación, el pivotamiento parcial (por filas) intercambia el elemento pivote, $a_{kk}^{(k)}$, por el elemento de *mayor magnitud* de la parte triangular inferior de la columna k-ésima de A_k, que denotamos como $a_{jk}^{(k)}$, con $j \geq k$. (Nótese que este criterio también gestiona el caso extremo en el que el elemento pivote es nulo.) Matemáticamente, esta operación equivale a multiplicar A_k por una matriz de permutación, P_{k+1}, que intercambia las filas k y j de A_k. Posteriormente, la transformación Gaussiana L_{k+1}^{-1} se aplica sobre la matriz permutada $P_{k+1}A_k$, y la eliminación continúa en la siguiente iteración con la matriz $A_{k+1} = L_{k+1}^{-1}P_{k+1}A_k$. Tras aplicar este proceso para $k = 1, \ldots, n-1$, se obtiene una factorización LU de la matriz

$$PA = \hat{A} = \hat{L}\hat{U} + \hat{E}, \tag{2.37}$$

en la que $P = P_{n-1} \cdots P_1$ es una matriz de permutación, $\hat{L} = \hat{L}_{n-1} \cdots \hat{L}_1$, con $\hat{L}_k = P_{n-1} \cdots P_{k+1}L_k$, es una matriz triangular inferior unidad, $\hat{U} = L_{n-1}^{-1}P_{n-1} \cdots L_1^{-1}P_1A$ es una matriz triangular superior, y $\hat{E}$ es una matriz de perturbación debida a los errores de redondeo. Además, se cumple que $\|\hat{L}\|_\infty \leq n$, aunque existen casos en los que $\|\hat{U}\|_\infty / \|\hat{A}\|_\infty \approx 2^{n-1}$. Dado que 2^{n-1} es muy grande incluso para valores modestos de n (por ejemplo, con $n = 100$, $2^{n-1} > 10^{29}$), no es posible afirmar que la eliminación Gaussiana con pivotamiento parcial sea, desde el punto de vista teórico, un proceso regresivamente estable. Sin embargo, años de experiencia con este método revelan que este tipo de situaciones (crecimiento arbitrario de los elementos de $\hat{U}$) es extremadamente infrecuente en la práctica. De hecho, el pivotamiento parcial es el método por defecto que se usa en paquetes de software numérico *fiable* como LAPACK [16].

En el caso de los métodos directos para matrices *dispersas* es necesario prestar mucha más atención al sobrecoste, en forma de número de operaciones y elementos no nulos adicionales, que puede suponer el uso de pivotamiento. En esta familia de métodos existe una fase previa a la factorización numérica en la que se emplean, entre otros, *algoritmos de ordenación para la reducción de llenado*. Estos algoritmos calculan heurísticamente una solución al problema de minimización de llenado, que consiste en obtener dos matrices de permutación, $\hat{P}$ y $\hat{Q}$, de tal forma que el llenado correspondiente a la factorización de $\hat{P}A\hat{Q}$ sea mínimo. Los intercambios de filas y/o columnas efectuados por la estrategia de pivotamiento para mejorar la estabilidad numérica pueden destruir la estructura de $\hat{P}A\hat{Q}$ y, como consecuencia, el llenado correspondiente a la factorización de $P\hat{P}A\hat{Q}$ puede ser significativamente más grande que el de $\hat{P}A\hat{Q}$. A pesar de que la solución óptima depende en gran medida de cada problema, el objetivo es lograr un *compromiso razonable* entre llenado y estabilidad numérica.

Existe un amplio abanico de soluciones para resolver el problema discutido en el párrafo anterior. En un extremo se encuentran las técnicas de *pivotamiento estático*, es decir, aquellas soluciones en las que no se permite variar la secuencia de elementos pivote *durante el proceso de factorización*. Estas técnicas son especialmente útiles cuando A es diagonal dominante ya que, para este tipo de matrices, no es necesario emplear pivotamiento [76]. Cuando A no es diagonal dominante, en algunos casos se puede recuperar parte de la precisión perdida por problemas de estabilidad regresiva empleando, después de calcular la factorización, un refinamiento iterativo [76]. En el marco del pivotamiento estático son especialmente importantes las técnicas descritas en [63, 64, 123], que tienen por objeto incrementar la magnitud de los elementos de la diagonal de A en relación con la magnitud de los elementos situados en la fila y columna correspondiente; es decir, aumentan el grado de dominancia de los elementos diagonales de la matriz. Para ello, en una fase previa a la factorización, denominada preproceso, se realizan dos transformaciones encadenadas del sistema de partida: en primer lugar, se calculan dos matrices diagonales D_1 y D_2, y se escala la matriz del sistema, es decir, $A \rightarrow D_1AD_2$; y en segundo lugar, se obtiene una matriz de permutación $\tilde{P}$, y se permutan las filas de D_1AD_2, es decir, $D_1AD_2 \rightarrow \tilde{P}D_1AD_2$. Posteriormente, se procede a la resolución del sistema obtenido

$$\tilde{P}D_1AD_2y = \tilde{P}D_1b, \quad x = D_2y. \tag{2.38}$$

Experimentalmente se ha observado que estas transformaciones del sistema de partida no sólo reducen en muchos casos la necesidad de emplear pivotamiento en el proceso de eliminación del sistema preprocesado (2.38) sino que, además, incrementan la precisión de la solución calculada [63, 64, 123]. Es necesario mencionar que, aunque en [123] se propusieron inicialmente una serie de heurísticos para obtener D_1, D_2 y $\tilde{P}$, éstos no explotaban la estructura dispersa de A. Los heurísticos propuestos en [123] junto con otros nuevos se desarrollaron e implementaron eficientemente para sistemas de ecuaciones lineales dispersos en [63, 64].

En el extremo contrario a las técnicas de pivotamiento estático, se encuentran las de *pivotamiento dinámico*, es decir, aquellas estrategias que tienen total libertad para elegir los elementos pivote durante la factorización, como por ejemplo los pivotamientos parcial y completo. En general, este tipo de técnicas no son prácticas en el caso disperso, por lo que se suele relajar el criterio que se emplea para elegir el elemento pivote en cada iteración del proceso de eliminación. Por ejemplo, en el pivotamiento completo con umbral, cualquiera de los elementos $a_{ij}^{(k)}$, con $i, j \geq k$, que cumplan que

$$|a_{ij}^{(k)}| \geq \alpha \max_{m,n \geq k} |a_{mn}^{(k)}|,$$

con $0 < \alpha \leq 1$ un parámetro prefijado, pueden ser escogidos en la iteración k-ésima como elemento pivote. Cuando α es igual a 1, entonces obtenemos el criterio correspondiente al pivotamiento completo. Cuando escogemos valores de α menores a 1, a cambio de un incremento potencial de la magnitud de los elementos de los factores, tenemos mayor libertad a la hora de elegir el elemento pivote. Esta libertad se puede utilizar para elegir aquel elemento pivote cuyas fila y columna produzcan menor llenado en la iteración actual del proceso de eliminación. Para esta última elección, se pueden emplear criterios de Markowitz [62].

En [76, 164] y [56, 62] aparecen más detalles sobre estrategias de pivotamiento en el contexto de métodos directos para matrices densas y dispersas, respectivamente.

Factorizaciones ILU

El objetivo fundamental de los precondicionadores es transformar el sistema de ecuaciones de partida en otro equivalente con propiedades más favorables con respecto al método iterativo de resolución escogido, en nuestro caso uno de los métodos basados en la generación de subespacios de Krylov. Estas propiedades favorables se suelen traducir en una reducción del número de iteraciones requeridas por el método de resolución para converger a la solución del sistema y, en aquellos casos en los que el incremento del coste por iteración se pueda amortizar con esta reducción en el número de iteraciones, una reducción del coste computacional asociado al resolutor. En el caso de los precondicionadores derivados a partir de factorizaciones ILU, la *experimentación* es la que revela, generalmente, hasta qué grado tienen éxito, fundamentalmente por el hecho de que las reglas de descarte que se emplean están basadas en criterios heurísticos, muy baratos computacionalmente, pero que no gestionan, al menos directamente y en el caso general, todos los factores de los que depende la convergencia del método iterativo. Esta situación se complica teniendo en cuenta que, para sistemas no simétricos generales, se desconoce cómo describir la convergencia de los métodos iterativos de Krylov empleando exclusivamente información "sencilla" derivada a partir de la matriz de coeficientes, como por ejemplo, su espectro de valores propios.

A pesar de los inconvenientes mencionados en el párrafo anterior, las factorizaciones ILU se han aplicado con éxito como precondicionadores de los sistemas de ecuaciones que surgen en multitud de aplicaciones académicas y del mundo real. La experimentación exhaustiva con esta familia de precondicionadores en el contexto de muy diversas aplicaciones, a lo largo de varias décadas, ha permitido entender, detectar y gestionar una serie de causas y problemas asociados que suelen comprometer la *existencia*, *precisión* y *estabilidad* de las factorizaciones ILU y, por tanto, su grado de éxito cuando actúan como precondicionadores. Estos problemas se pueden clasificar de la forma siguiente:

- *Imprecisión causada por elementos pivote muy pequeños.*

  Los elementos pivote de una factorización ILU pueden ser arbitrariamente pequeños, incluso nulos. Con frecuencia, este factor causa que la factorización ILU sea inestable, es decir, una factorización en la que los errores de redondeo afectan de forma significativa a la precisión de la factorización. Las técnicas de pivotamiento que se emplean en los métodos directos para controlar la magnitud de los elementos de los factores no siempre son necesarias (e incluso deseables por cuestiones de eficiencia computacional) en el caso de las factorizaciones ILU. Para justificar, en parte, esta afirmación, hay que analizar el crecimiento de la magnitud de los elementos de L y de U, que depende, entre otros factores, de la frecuencia con la que se actualiza cada elemento. En las factorizaciones ILU, el crecimiento de $\|\tilde{L}\|$ y de $\|\tilde{U}\|$ es más moderado, ya que, al descartar elementos no nulos durante el proceso, el número de actualizaciones de cada elemento se reduce. En [50] aparece una descripción más detallada de las razones por las cuales las técnicas de pivotamiento que se emplean en los métodos directos no se usan con tanta frecuencia en las factorizaciones ILU. De hecho, muchas de las técnicas básicas presentadas del Apartado 2.1.1 se formularon originalmente sin pivotamiento, precondicionando de forma exitosa los sistemas de las aplicaciones seleccionadas para su evaluación. Aún así, es necesario incorporar mecanismos para gestionar los elementos pivote muy pequeños o nulos.

- *"Inestabilidad" en la aplicación del precondicionador.*

  En el contexto de las factorizaciones ILU se habla de "inestabilidad" en la aplicación del precondicionador para referirse a aquellas situaciones en las que la norma de la inversa de los factores triangulares $\|\tilde{L}^{-1}\|$ y/o $\|\tilde{U}^{-1}\|$, es extremadamente grande. Para sistemas no simétricos y/o altamente indefinidos, a menudo se observa que los factores incompletos resultantes de las factorizaciones ILU básicas, como por ejemplo la ILU(l) o la ILUT(τ, p), están muy mal condicionados, incluso cuando la matriz de coeficientes del sistema está bien condicionada [27]. Como consecuencia, en la aplicación del precondicionador, que involucra la resolución de sendos sistemas triangulares con $\tilde{L}$ y $\tilde{U}$, se pueden producir errores de magnitud significativa, destruyendo por completo el rendimiento de la factorización ILU como precondicionador [158]. Además, como efecto complementario, si $\tilde{L}^{-1}$ y/o $\tilde{U}^{-1}$ tienen entradas muy grandes, entonces en la expresión $A = \tilde{L}\tilde{U} + R$ (ver (2.19)), es probable que la matriz del sistema precondicionado sea muy diferente de la matriz identidad, es decir, $\tilde{L}^{-1}R\tilde{U}^{-1}\tilde{D}^{-1}$ tendrá probablemente entradas muy grandes. Este problema fue inicialmente detectado en [158], y analizado en detalle en [67] en el contexto de la discretización mediante diferencias finitas de ecuaciones diferenciales elípticas con derivadas parciales de primer orden. En [50], este problema se analizó experimentalmente para un amplio abanico de aplicaciones seleccionadas de las colecciones de matrices de Harwell-Boeing, UMFPACK y SPARSKIT y, en [27], en el contexto de la EDP de convección–difusión. Como se menciona en [50], la inestabilidad en la aplicación del precondicionador parece estar asociada al efecto que tiene el descarte de elementos no nulos en el proceso de factorización aproximada. Cabe destacar que el uso de elementos pivote muy pequeños en el proceso de factorización automáticamente da lugar a inestabilidad en la aplicación del precondicionador. Sin embargo, este problema también puede surgir cuando no se emplean elementos pivote muy pequeños [27, 158].

  Para identificar la inestabilidad en la aplicación del precondicionador se puede inspeccionar, después de calcular la factorización, el valor de $\|M^{-1}z\|_\infty$, con $M = \tilde{L}\tilde{U}$ el precondicionador derivado, y $z = (1 \dots 1)^T$. A pesar de que esta comprobación únicamente proporciona una cota inferior de la normal matricial infinito de M^{-1}, ya que $\|M^{-1}\|_\infty\|z\|_\infty = \|M^{-1}\|_\infty \geq \|M^{-1}z\|_\infty$, resulta computacionalmente económica y, además, en la práctica es útil, especialmente cuando $\|M^{-1}z\|_\infty$ tiene un valor muy grande. Por ejemplo, un valor del orden de $1/u$ o superior indica que la factorización ILU calculada no resultará útil para

acelerar la convergencia del método iterativo [50]. En [27] se propuso evaluar directamente la desviación de la matriz del sistema precondicionado con respecto a la matriz identidad empleando la norma Frobenius, es decir, $\|I-(\tilde{L}\tilde{U})^{-1}A\|_F = \|(\tilde{L}\tilde{U})^{-1}R\|_F$, con R la matriz residuo de la factorización ILU. Esta métrica, aunque no tan económica computacionalmente como la anterior, mide directamente la magnitud de la matriz de perturbación que causa el descarte de elementos en la matriz del sistema precondicionado.

El problema de la inestabilidad en la aplicación del precondicionador se puede atenuar, hasta cierto punto, perturbando los elementos de la diagonal principal de A antes de calcular la factorización ILU. Por ejemplo, en [158] se desarrollaron heurísticos para perturbar la diagonal principal de A con el objeto de aumentar el peso relativo de la diagonal principal de los factores incompletos. Estos heurísticos dieron muy buen resultado en el contexto de los sistemas no simétricos resultantes de la discretización mediante diferencias finitas de ciertas EDPs elípticas. Nótese que, si $\tilde{L}$ ($\tilde{U}$) es una matriz (estrictamente) diagonal dominante, entonces es posible determinar a priori una cota superior de la norma de $\tilde{L}^{-1}$ ($\tilde{U}^{-1}$) [94]. Otro tipo de técnicas perturban los elementos pivote dinámicamente durante la factorización. Por ejemplo, si la magnitud de un elemento pivote es más pequeña que un umbral preestablecido, entonces su valor se reemplaza por el de este umbral, manteniendo el signo original del elemento pivote. En el estudio experimental realizado en [50] se observó que, para ciertas matrices, existe una correlación entre el aumento de este umbral y la disminución de $\|M^{-1}z\|_\infty$. Sin embargo, el umbral óptimo en términos de la calidad de la factorización ILU como precondicionador no suele ser el que proporciona el menor valor de $\|M^{-1}z\|_\infty$, dado que las perturbaciones disminuyen la precisión de la factorización. Cabe destacar que las técnicas de perturbación (de la diagonal principal de A o de los elementos pivote durante la factorización) también se han empleado extensivamente en el caso SDP para evitar elementos pivote negativos, nulos o muy pequeños. Cuando surgen elementos pivote negativos, la factorización incompleta de Cholesky falla al calcular la raíz de estos valores, y la factorización incompleta LDL^T acaba derivando un precondicionador que no es SDP. Como consecuencia, el sistema precondicionado (1.4) no es SDP. (Nótese que métodos iterativos como el PCG requieren que (1.4) sea SDP para estar definidos apropiadamente.) El problema de esta familia de técnicas radica en la elección de un valor apropiado para los parámetros de los que depende la magnitud de las perturbaciones. Por ejemplo, un umbral muy grande puede introducir perturbaciones muy grandes y, por tanto, imprecisión causada por estas perturbaciones. Por el contrario, si este umbral se escoge demasiado pequeño, entonces pueden aparecer elementos pivote muy pequeños, causando inestabilidad en la factorización y en la aplicación del precondicionador. El equilibrio entre estabilidad y precisión no siempre se puede lograr, siendo necesario emplear otro tipo de estrategias en tales casos. Para más detalles sobre este tipo de técnicas, consultar [18, 68, 99, 107, 114, 121, 158].

- *Imprecisión causada por el descarte de elementos.*

  La factorización LU se transforma en incompleta descartando elementos durante el proceso en base a los criterios definidos al diseñar las reglas de descarte. Cada uno de estos elementos contribuye a la matriz de error R en la expresión (2.12) y, por tanto, el descarte de elementos influye sobre la precisión de la factorización. Cuando una factorización ILU no es eficaz para acelerar la convergencia del método iterativo, inmediatamente se identifica como posible causa el que se han descartado demasiados elementos durante el proceso, es decir, la factorización es imprecisa. Con las técnicas presentadas en el Apartado 2.1.1, la precisión de la factorización se puede aumentar admitiendo más llenado. Sin embargo, si la imprecisión no se debe exclusivamente al descarte de elementos, sino que además han aparecido elementos pivote muy pequeños en el proceso y una factorización inestable, entonces es posible que incrementar exclusivamente el llenado no ayude. Además, es necesario

tener en cuenta el efecto que puede tener esta solución sobre la estabilidad en la aplicación del precondicionador. En [158] se analizó el efecto que tiene admitir progresivamente más llenado sobre la estabilidad en la aplicación del precondicionador en el contexto de los sistemas no simétricos que surgen en la discretización de ciertas EDPs elípticas. La conclusión principal que se derivó en este trabajo es que, *para este tipo de problemas*, admitir progresivamente más llenado no sólo no ayuda, sino que hace más severo el problema. Esta tendencia se mantiene hasta que se admite suficiente llenado como para que la precisión de la factorización ILU sea del mismo orden que la de la factorización LU. Nótese que en este último caso se obtiene un precondicionador muy costoso de calcular y aplicar y, por tanto, inútil a nivel práctico.

- *Elementos pivote nulos.*

  La factorización ILU falla si alguno de los elementos pivote, es decir, $\hat{a}_{kk}^{(k-1)}$, para $k = 1, \ldots, n$, es nulo. En este caso el precondicionador derivado es singular y, por tanto, no es invertible, es decir, *no se puede emplear como precondicionador.*

  La existencia de una factorización ILU que no use pivotamiento, ni descarte elementos que pertenezcan a la diagonal principal, *está garantizada*, en aritmética exacta, si A es una M-matriz (consultar la definición en la Sección 1.7). Este resultado se demostró originalmente en [118] para la factorización ILU$_{\mathcal{P}}$, aunque se puede extender fácilmente para cualquier otro tipo de factorización ILU [138]. Para este tipo de matrices, los elementos pivote cumplen que $\hat{a}_{kk}^{(k-1)} > 0$, para $k = 1, \ldots, n$, y, en consecuencia, la factorización (2.12) se completa sin divisiones por cero, es decir, $M = \tilde{L}\tilde{U}$ es no singular. Además, si A es SDP, entonces la factorización incompleta $A = \tilde{L}\tilde{D}\tilde{L}^T + R$ obtiene un precondicionador $M = \tilde{L}\tilde{D}\tilde{L}^T$ que es SDP, ya que todos los elementos de la diagonal principal de $\tilde{D}$ son positivos. Esta propiedad del precondicionador es necesaria para métodos iterativos como el PCG, que requieren que la matriz del sistema precondicionado (1.4) sea SDP para estar definidos apropiadamente. El resultado correspondiente a las M-matrices se extendió posteriormente en varios trabajos [162, 114] a un tipo más extenso de matrices, el de las H-matrices. Cabe destacar, por una parte, que las H-matrices incluyen las matrices diagonal dominantes y, por otra, que las M-matrices y H-matrices surgen en aplicaciones de cierta importancia, como la resolución numérica de ciertas EDPs elípticas.

  Desafortunadamente, en la gran mayoría de las aplicaciones no surgen estos tipos de matrices. Además, los resultados teóricos correspondientes a las M-matrices y H-matrices no aportan garantías sobre la estabilidad numérica del proceso de eliminación aproximada. En el proceso de factorización ILU de una matriz general, los elementos pivote nulos aparecen por las mismas causas que en el de la eliminación Gaussiana [76, 164], aunque en el caso de las factorizaciones ILU es necesario tener en cuenta las perturbaciones ocasionadas al descartar elementos en el proceso de factorización. Por ejemplo, en aritmética exacta, un elemento pivote nulo surge si $A - R$ tiene algún menor principal singular, *independientemente de que A admita una factorización LU o no.* En este caso, las perturbaciones ocasionadas al descartar elementos tienen el efecto de transformar algún menor principal de A en singular. El mismo razonamiento se puede aplicar para el resto de las causas que originan elementos pivote nulos en la eliminación Gaussiana. Por ejemplo, si A está bien condicionada, pero como resultado de descartar elementos, $A - R$ está muy mal condicionada, entonces es probable que surjan elementos pivote nulos, esta vez por el efecto combinado del descarte de elementos y de los errores de redondeo.

  En el caso general, no existe una caracterización definitiva para predecir si A admitirá o no una factorización ILU. Sin embargo, los elementos pivote nulos surgen con mucha frecuencia en matrices con estructura muy irregular y elementos nulos en la diagonal principal. La estructura u ordenación de estas matrices favorece que no se cree llenado en la posición

correspondiente a un elemento de la diagonal principal que es nulo. Por ejemplo, suponiendo que la matriz de coeficientes cumple que $a_{jk} = a_{kj} = 0$, para $j = 1, \ldots, k$, entonces la factorización ILU(0) falla, ya que el elemento pivote correspondiente a la iteración k-ésima del proceso de eliminación aproximada, es decir, $\hat{a}_{kk}^{(k-1)} = a_{kk} - \sum_{j=1}^{k-1} \tilde{l}_{kj}\tilde{u}_{jk}$, es nulo. La estrategia más efectiva en estas situaciones consiste en técnicas de pivotamiento dinámico, de forma similar a los métodos directos para matrices dispersas. El pivotamiento cambia dinámicamente la estructura de la matriz, gestionando directamente el problema. Por ejemplo, en [135] se desarrolló la factorización ILUTP(τ,p), que incorpora pivotamiento parcial por columnas a la factorización ILUT(τ,p). Es necesario recalcar que el pivotamiento, aunque muy efectivo en la práctica, no resuelve el problema en todos los casos. En [135] se demostró que, incluso en aritmética exacta, la factorización ILUTP(τ,p) puede quedarse atrapada y fallar; es decir, en una iteración concreta del proceso de eliminación, puede resultar imposible escoger un elemento pivote no nulo a través de intercambios de columnas. Este resultado se puede aplicar también a otros tipos de pivotamiento, como el pivotamiento parcial por filas, o el pivotamiento por columnas.

Los problemas anteriores pueden surgir simultáneamente y, en consecuencia, unos problemas pueden enmascarar a otros. Por ejemplo, la imprecisión causada al descartar elementos puede dar lugar a elementos pivote muy pequeños. Estos elementos pivote pueden a su vez causar una factorización inestable y un elemento pivote nulo por el efecto de los errores de redondeo. Cuando la factorización falla, puede resultar complejo identificar los problemas causantes. Además, estos problemas pueden interactuar de forma muy compleja. Por ejemplo, al admitir más llenado para aumentar la precisión de la factorización, en la nueva factorización pueden surgir elementos pivote más pequeños, y estos elementos pivote pueden a su vez causar inestabilidad en la aplicación del precondicionador. Es necesario recalcar que, experimentalmente, se observa que estos problemas surgen con más frecuencia y con mayor impacto en matrices que no tienen propiedades favorables como la simetría, ser diagonal dominante, ser definida positiva, etc. y, por tanto, las tasas de fallos más altas se dan a menudo para matrices no simétricas (estructural y numéricamente), muy poco estructuradas e indefinidas. Sin embargo, aunque esta caracterización es útil, la experimentación es la que revela finalmente en qué grado se dan estos problemas, y cómo afectan a la calidad de la factorización como precondicionador. Estos problemas, entre otros, complican extraordinariamente el desarrollo de resolutores de propósito general que sean robustos en todas las situaciones. De hecho, las factorizaciones ILU fallan en muchas situaciones, aunque su grado de éxito se puede incrementar de forma significativa teniendo en cuenta las propiedades del problema concreto y las causas de los problemas que se encuentra la factorización y/o la posterior resolución del sistema precondicionado. La esperanza es que, para una clase de problemas concreta, se pueda encontrar una estrategia, o un conjunto de parámetros, que sea efectiva para todas los problemas de esa clase, aunque con frecuencia se observa que es necesario tener en cuenta las características específicas de cada sistema.

Las técnicas desarrolladas por Bollhöfer y colaboradores en varios trabajos [34, 32, 33, 30, 141, 31], han permitido mejorar de forma significativa la robustez, fiabilidad y eficiencia de la factorizaciones ILU cuando actúan como precondicionadores, especialmente para matrices no simétricas y/o indefinidas. La clave de las técnicas numéricas desarrolladas en estos trabajos es la monitorización y control, durante el proceso de factorización ILU, del crecimiento de la magnitud de la inversa de los factores aproximados, es decir, de la estabilidad en la aplicación del precondicionador. En las reglas de descarte basadas en la inversa (ver Apartado 2.1.1), la magnitud de las entradas de $\tilde{L}^{-1}$ y $\tilde{U}^{-1}$ (estimada a través de los estimadores del número de condición) influye sobre la magnitud del umbral empleado para descartar o admitir elementos no nulos de los factores aproximados. En [29, 30] las técnicas de descarte basadas en la inversa se combinaron con técnicas de pivotamiento (dinámico) adoptadas del área de los métodos directos para matrices dispersas. En este caso, el uso de pivotamiento se debe, entre otros factores, a que la magnitud del umbral que se emplea para admitir o descartar elementos en este tipo

de reglas de descarte depende del crecimiento de la magnitud de las entradas de la inversa de los factores incompletos. El uso de elementos pivote muy pequeños inmediatamente da lugar a entradas en $\tilde{L}^{-1}$ y $\tilde{U}^{-1}$ muy grandes y, por tanto, sin pivotamiento existe el peligro potencial de que el método admita mucho llenado. Este argumento sirve para ilustrar que, en este tipo de factorizaciones ILU, las reglas de descarte basadas en la inversa únicamente son útiles a nivel práctico si existe un mecanismo para controlar la magnitud de las entradas de $\tilde{L}^{-1}$ y $\tilde{U}^{-1}$. En la biblioteca ILUPACK, los estimadores del número de condición (ver Algoritmo 2.11) se incorporan a las técnicas de pivotamiento, controlando directamente la magnitud de las entradas de $\tilde{L}^{-1}$ y $\tilde{U}^{-1}$. Para más detalles, se puede consultar [34, 141, 31], y la Sección 2.2.

2.1.3. Efecto de las reordenaciones

Las factorizaciones ILU son sensibles a variaciones en la ordenación (numeración) de las incógnitas y ecuaciones del sistema. Estos cambios, que se llevan a cabo través de permutaciones de sus filas y/o columnas, se denominan *reordenaciones*. En el contexto de las técnicas de precondicionado que nos ocupan, este término se refiere, generalmente, a las permutaciones que se obtienen y aplican antes de calcular la factorización ILU. Las reordenaciones se han empleado para reducir el llenado (como en los métodos directos para matrices dispersas), *aumentar el grado de concurrencia en la construcción y aplicación del precondicionador*, y mejorar la estabilidad del proceso de factorización.

Si P^T y Q son dos matrices de permutación, entonces el sistema (1.1) se puede reemplazar por el sistema equivalente

$$P^T AQy = P^T b, \quad x = Qy. \tag{2.39}$$

La solución del sistema original se obtiene aplicando un método iterativo al sistema que resulta de precondicionar (2.39) mediante una factorización ILU de $P^T AQ$. Formalmente, $P^T AQ$ y A son matrices distintas. Sin embargo, es común referirse a $P^T AQ$ empleando el término "ordenación o reordenación de A". Un tipo concreto de reordenaciones de A son las correspondientes a permutaciones simétricas, es decir, $Q = P^T$. En este caso, la reordenación $P^T AP$ tiene el mismo espectro de valores propios que A, el mismo conjunto de elementos en la diagonal principal (aunque situados en posiciones distintas) y, además, es estructuralmente simétrica si A también lo es. Los heurísticos más comunes para calcular permutaciones simétricas a partir de la estructura de A requieren que A sea estructuralmente simétrica. Para matrices que no lo son, se suele emplear la estructura de la matriz simétrica $A + A^T$ para superar esta limitación.

Las reordenaciones afectan a la velocidad de convergencia de los métodos iterativos precondicionados mediante factorizaciones ILU. El efecto de las reordenaciones ha sido estudiado, experimentalmente, en muchos trabajos. En general, estas dos variables (ordenación de A y convergencia) interactúan de forma compleja e impredecible, hasta tal punto que estos estudios no proporcionan explicaciones concluyentes a muchos de los fenómenos observados. En este apartado revisamos aquellas afirmaciones para las cuales existe cierto grado de consenso, centrándonos en las estrategias de reordenación para reducción de llenado que se desarrollaron originalmente en el contexto de los métodos directos para matrices dispersas [56, 62].

Para los métodos directos existen tres familias básicas de heurísticos para obtener soluciones razonables al problema del llenado mínimo: la reducción de ancho de banda y perfil, como por ejemplo, el algoritmo Cuthill-Mckee Inverso (RCM) [56]; el algoritmo del mínimo grado [75] y sus variantes, como el mínimo grado aproximado (AMD) [14]; y el algoritmo de disección anidada [74] y sus variantes, como la disección anidada multinivel (MLND) [102]. Todos estos algoritmos calculan permutaciones simétricas y, además, se basan *exclusivamente* en la estructura (grafo de adyacencia) de la matriz de coeficientes; en otras palabras, la magnitud de las entradas no influye sobre las decisiones adoptadas al calcular P. Sin embargo, la eficacia de las factorizaciones ILU depende fuertemente, entre otros factores, de la magnitud de las entradas descartadas en el proceso de factorización. Por ello, no resulta sorprendente que métodos de reordenación de reducción de llenado de alta calidad para los métodos directos, como por ejemplo

AMD o MLND, puedan resultar en factorizaciones ILU de peor calidad que las obtenidas, por ejemplo, con la ordenación de partida. Para ilustrar este fenómeno, en la Figura 2.5 mostramos el patrón de elementos no nulos de la matriz residuo R (columna derecha) resultante de aplicar la factorización de Cholesky incompleta IC(0) a cuatro ordenaciones (columna izquierda) de una matriz de coeficientes de ejemplo. Esta matriz surge de la discretización, mediante diferencias finitas centradas de segundo orden, del operador diferencial de Laplace en una malla 2D de 30×30 nodos, y da lugar a una matriz SDP con 900 ecuaciones e incógnitas y 5 elementos no nulos por cada fila. La ordenación natural (primera fila de la Figura 2.5) se obtiene al numerar los nodos de la malla de forma secuencial, por filas, y las ordenaciones MLND, RCM y AMD, al aplicar estos algoritmos de reordenación a la misma. Es necesario recalcar que la factorización IC(0) no se encuentra con elementos pivote nulos o negativos y, por tanto, el precondicionador derivado es SDP. Además, tampoco se dan problemas de estabilidad en la factorización, ni en la aplicación del precondicionador. Como se puede observar en la parte inferior de cada diagrama, la norma Frobenius de la matriz residuo, $\|R\|_F$, es equivalente para la ordenación natural y RCM, y aproximadamente igual para MLND y AMD. Sin embargo, la norma de estas dos últimas ordenaciones dobla a la de las dos primeras. El número de iteraciones requeridas por el PCG para resolver el sistema precondicionado correspondiente a las cuatro ordenaciones, es de 22, 39, 22 y 41, para la ordenación natural, MLND, RCM y AMD, respectivamente. Por lo tanto, para este problema y resolutor, basado en el binomio IC(0)-PCG, el uso de algoritmos de reordenación prestados del área de los métodos directos no mejora el coste computacional, e incluso puede empeorarlo. Nótese que la factorización IC(0) construye un factor incompleto de Cholesky, $\tilde{L}$, con el mismo número de elementos no nulos que A, independientemente de la ordenación empleada y, en consecuencia, no es descabellado afirmar que el coste computacional por iteración será el mismo para todas las opciones.

El fenómeno discutido en el párrafo anterior fue observado y estudiado en detalle en [65] para los sistemas SDP que surgen en la resolución de varias EDPs bidimensionales aplicadas sobre dominios computacionales regulares, como el cuadrado unidad. Además de la ecuación de Laplace con condiciones de contorno de Dirichlet, cuya discretización da lugar a matrices como las discutidas en el párrafo anterior, en este trabajo se consideraron varias modificaciones de la EDP básica, como la presencia de anisotropía en la EDP (con coeficientes distintos para cada derivada parcial segunda), o la presencia de coeficientes discontinuos (con valor variable en función de la región del dominio). Estas modificaciones afectan a la magnitud de los elementos de A, pero no a su estructura, que depende del tipo de diferencia finita escogida para aproximar la ecuación diferencial en cada punto de la malla. Además, se consideró el efecto de las reordenaciones sobre varios resolutores basados en el binomio factorización IC-PCG, incluyendo IC(0), IC(1) y una variante de la factorización incompleta de Cholesky con descarte en base a la magnitud de los elementos de los factores. Experimentalmente se observó que, para este tipo de problemas, la velocidad de convergencia del resolutor IC(0)-PCG es significativamente menor cuando se emplea el algoritmo del mínimo grado en lugar de la ordenación natural. La explicación que se dio en [65] emplea el tamaño medio de los elementos descartados: para la ordenación calculada por el algoritmo del mínimo grado, el tamaño medio de los elementos descartados es significativamente mayor que en la ordenación natural, de tal forma que la factorización IC(0) resultante es menos precisa en el primer caso. Sin embargo, una afirmación importante es que, en general, los métodos de reducción de llenado ofrecen un rendimiento mucho mejor en comparación con la ordenación de partida si se admite cierto grado de llenado durante el proceso de factorización ILU. Como se demostró en [65], para la factorización IC(1) y la basada en descarte con umbral absoluto $\tau = 10^{-2}$, el mérito relativo de las distintas estrategias de reordenación cambia hasta tal punto que los algoritmos del mínimo grado y de la disección anidada, entre otros, ofrecen un rendimiento comparable e incluso mejor que la ordenación natural en ciertas situaciones, es decir, resuelven el sistema en menos tiempo. Sin embargo, la impresión general es que, *para este tipo de problemas*, en los que las factorizaciones ILU sin o con poco llenado

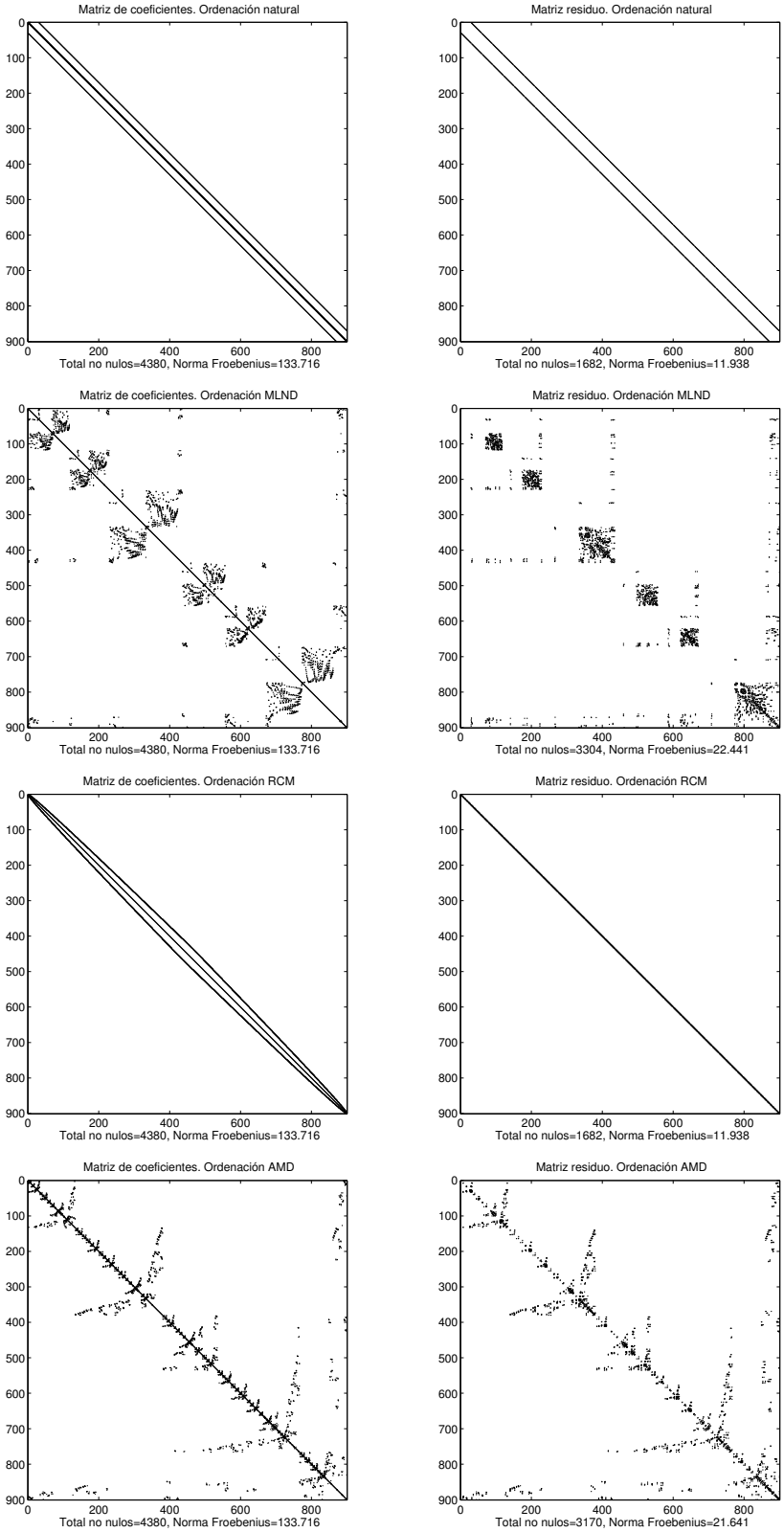


Figura 2.5: Patrón de elementos no nulos de la matriz de coeficientes (izquierda) y matriz residuo (derecha) de la factorización IC(0) correspondiente a cuatro ordenaciones distintas. De arriba a abajo: ordenación natural, y ordenaciones resultantes de aplicar los algoritmos MLND, RCM, y AMD. La parte inferior de cada diagrama muestra el número de elementos no nulos y la norma Frobenius correspondiente a la matriz representada.

proporcionan soluciones muy competitivas, no hay mucho que ganar mediante el uso de algoritmos de reducción de llenado. Para matrices que surgen de la discretización mediante el método de los elementos finitos, en las que un orden "natural" no siempre existe, los autores de [65] recomiendan el uso de RCM. En varios trabajos [52, 165, 23], el uso de ciertas variantes de RCM resultó beneficioso para problemas que surgen en las simulaciones de las reservas de petróleo. Una explicación intuitiva del efecto beneficioso que suele tener el método RCM sobre resolutores del tipo IC-PCG puede encontrarse en [42].

El panorama es diferente para problemas que dan lugar a matrices no simétricas e/o indefinidas. En este caso, las reordenaciones (de reducción de llenado o de otro tipo) pueden mejorar significativamente la velocidad de los métodos de Krylov precondicionados mediante factorizaciones ILU. Esta afirmación fue respaldada en [66], trabajo en el que se estudió el efecto de varias estrategias de reordenación sobre el rendimiento del resolutor basado en el binomio ILU(0)-GMRES en el contexto de ecuación compresible de Navier-Stokes sobre mallas no estructuradas. En varios estudios sistemáticos [25, 27], se observó que el algoritmo RCM ofrece los mejores resultados en muchos casos, especialmente cuando se combina con la factorización ILUT en el contexto de la discretización mediante diferencias finitas de la ecuación de convección-difusión. Este tipo de problemas da lugar a matrices *no simétricas*, aunque con la misma estructura (simétrica) que la matriz considerada en la Figura 2.5. Para estos problemas, RCM también obtuvo con frecuencia los mejores resultados en términos de robustez y fiabilidad, es decir, con RCM, el resolutor ILU-GMRES falló un menor número de veces. En [27] se observó que, en algunos casos, algoritmos de reducción llenado como el mínimo grado o el RCM, pueden tener un efecto positivo sobre la estabilización de la aplicación del precondicionador. De hecho, en algunos casos, resultó necesario emplearlos (aunque no siempre suficiente), ya que la factorización ILU con la ordenación natural daba lugar a factores incompletos muy mal condicionados y, por tanto, a un precondicionador inútil.

Un avance reciente y *significativo* en materia de estrategias de preprocesado para sistemas no simétricos y/o indefinidos es el desarrollo de algoritmos de reordenación (y escalado) que consideran la magnitud de los elementos de A con el objeto de aumentar, en la matriz permutada, el peso (magnitud) relativo de la diagonal principal [63, 64, 123, 31, 141]. Los heurísticos considerados en [63, 64, 123] dan lugar a permutaciones no simétricas, con $Q = I$, aunque en [31, 141] se presentó una versión de estos heurísticos que obtiene permutaciones simétricas. Mientras que las permutaciones simétricas mantienen los elementos de la diagonal principal de A en la matriz permutada, $P^T AP$, en su lugar, el heurístico propuesto en [31, 141] aumenta el peso relativo de la diagonal principal formada por bloques de dimensión 2×2. Esta propiedad resulta útil para mejorar la estabilidad numérica de los algoritmos de factorización de matrices simétricas indefinidas [76]. Como revisamos en el Apartado 2.1.2, los heurísticos que aumentan el grado de dominancia de los elementos diagonales de la matriz del sistema se emplean como estrategias de pivotamiento estático en los métodos directos, aunque en [63, 64] ya se advirtió y observó que podían resultar muy beneficiosas para mejorar la robustez y fiabilidad de los resolutores basados en la generación de subespacios de Krylov precondicionados mediante factorizaciones ILU. Un estudio sistemático de estas estrategias de reordenación y escalado, en el contexto de múltiples aplicaciones, provenientes por ejemplo de ingeniería química, economía, análisis de circuitos, y dinámica de fluidos computacional, puede encontrarse en [24]. En este estudio, el escalado, $A \rightarrow D_1 AD_2$, y la permutación no simétrica, $D_1 AD_2 \rightarrow PD_1 AD_2 = \hat{A}$, empleados para aumentar el peso relativo de la diagonal principal, se combinó con una permutación simétrica, $\hat{A} \rightarrow \hat{P}\hat{A}\hat{P}^T$, obtenida como resultado de aplicar una estrategia de reducción de llenado a la estructura de la matriz simetrizada $\hat{A} + \hat{A}^T$. Los experimentos efectuados en éste y otros trabajos [31, 141] indican que el uso de estas estrategias, permite calcular factorizaciones estables y eficaces en muchas situaciones, especialmente para matrices con estructura muy irregular y elementos muy pequeños o nulos en la diagonal principal de A [24].

2.1.4. Métodos multinivel

En el contexto de la resolución numérica de ecuaciones diferenciales en derivadas parciales, la velocidad de convergencia de los métodos iterativos basados en la generación de subespacios de Krylov tiende a degradarse de forma *significativa* a medida que crece la dimensión de los sistemas de ecuaciones. Por lo tanto, al aumentar el tamaño del problema, son necesarias muchas más iteraciones para cumplir con el criterio de convergencia escogido a la hora de diseñar el resolutor. Además, para alguno de los métodos que se emplean para resolver sistemas no simétricos, como el GMRES [138], el coste (en número de operaciones) por iteración y la memoria consumida para resolver un sistema dado no son constantes, y aumentan con cada iteración. En general, el aumento del coste por iteración y el incremento del número de iteraciones son dos factores que limitan la eficiencia de estos métodos para problemas de *gran dimensión*. Por esta razón, los métodos iterativos basados en la generación de subespacios de Krylov son considerados como métodos *no escalables*.

Para muchos tipos de EDPs elípticas de segundo orden en dos o tres dimensiones, cuya discretización, ya sea mediante elementos o diferencias finitas, da lugar a matrices SDP, es posible *cuantificar* a qué ritmo se degrada la velocidad de convergencia cuando se utiliza el método PCG como resolutor. Para este tipo de problemas se cumple que el número de condición espectral de A crece como una función de $O(h^{-2})$ cuando $h \to 0$, donde $0 < h \leq 1$ es un parámetro que depende de las propiedades (tamaño, geometría, etc.) de la malla empleada para discretizar el dominio computacional de la EDP. Por ejemplo, suponiendo que se emplea una malla cuadrada o cúbica de nodos para discretizar el dominio de la EDP, entonces h es simplemente la distancia que hay entre dos puntos consecutivos de la malla. Por otra parte, la velocidad de convergencia del método CG (sin precondicionar) depende del número de condición espectral de A, $\kappa(A)$, a través de la siguiente relación

$$\frac{\|e^{(k)}\|_A}{\|e^{(0)}\|_A} \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^k, \tag{2.40}$$

donde $e^{(k)} = x - x^{(k)}$ es el vector de error correspondiente a la aproximación obtenida tras k iteraciones, $x^{(k)}$, y $x = A^{-1}b$. Por lo tanto, *el número de iteraciones crece como una función de orden* $O(h^{-1})$ *cuando* $h \to 0$.

La degradación de la velocidad de convergencia provocada al aumentar el tamaño de problema se puede atenuar empleando precondicionadores derivados a partir de factorizaciones ILU básicas, como la ILU(l) o la ILUT(τ, p). Sin embargo, la velocidad de convergencia de los métodos basados en la generación de subespacios Krylov precondicionados mediante factorizaciones ILU sigue degradándose, aunque de forma más moderada, a medida que crece la dimensión del sistema. Por ejemplo, para la resolución de EDPs elípticas de segundo orden en dos o tres dimensiones cuya discretización da lugar a matrices SDP, se puede emplear la factorización incompleta de Cholesky IC(0). El coste de aplicar este tipo de precondicionadores, es decir, el número de operaciones requeridas para resolver el sistema $\tilde{L}\tilde{L}^T z = r$, es del mismo orden que realizar un producto matriz-vector con A y, por tanto, el coste por iteración del método PCG sólo se incrementa ligeramente. Además, este incremento del coste por iteración se puede amortizar de sobra con la reducción en el número de iteraciones. Sin embargo, las factorizaciones IC(0) *no alteran la dependencia del número de condición espectral con respecto a* h; es decir, el número de condición espectral de la matriz del sistema precondicionado, $\tilde{L}^{-1}A\tilde{L}^{-T}$, sigue creciendo como un función de $O(h^{-2})$ cuando $h \to 0$. Es necesario remarcar que, aunque el orden de la función es equivalente para el sistema con y sin precondicionador, la función concreta de la que depende el número de condición está multiplicada por una constante mucho más pequeña en el caso del sistema precondicionado, hecho que se manifiesta con la reducción en el número de iteraciones para el resolutor IC(0)-PCG. Al incrementar el número de elementos no nulos de $\tilde{L}$ en relación a los de A, es decir, al admitir más llenado, el efecto del precondicionador obtenido

sobre la reducción del número de condición es mayor, pero la velocidad de convergencia sigue degradándose a medida que crece la dimensión del sistema. Este comportamiento es típico para sistemas que surgen de la discretización de EDPs.

Los problemas de escalabilidad de los métodos basados en la generación de subespacios de Krylov, motivaron el desarrollo de una familia de métodos iterativos alternativos, conocidos como métodos multimalla (*multigrid*). A diferencia de los métodos de Krylov, que son métodos de propósito general, los métodos multimalla fueron diseñados originalmente con el objetivo específico de resolver sistemas de ecuaciones que surgen en aplicaciones gobernadas por EDPs elípticas. Estos métodos convergen muy rápidamente a la solución del sistema y, además, para diversos problemas, *la velocidad de convergencia es independiente del valor de h*. El núcleo de estos métodos está basado en el empleo de una jerarquía de mallas de distintos tamaños. Por ejemplo, en el contexto de la EDP de Laplace en el cuadrado unidad, discretizada mediante una malla de 256×256 nodos, un resolutor multimalla podría considerar, además, mallas de tamaño 128×128, 64×64, 32×32, 16×16, y 8×8. La malla de mayor tamaño se denomina malla fina (*fine grid*); la de menor tamaño malla gruesa (*coarse grid)*; y si una malla es de mayor o menor tamaño que otra, se utilizan los términos "más fina" (*finer grid*) o "más gruesa" (*coarser grid*), respectivamente. Otro bloque básico de este tipo de métodos son los suavizadores (*smoothers*). Un suavizador es un método iterativo subsidiario, computacionalmente económico, que permite atenuar significativa y rápidamente, es decir, en pocas iteraciones, ciertas componentes espectrales del vector de error (o residuo). Los suavizadores que se emplean con mayor frecuencia se derivan a partir de métodos iterativos de punto fijo, como por ejemplo la iteración de Richardson, el método de Jacobi o el método de Gauss-Seidel. Aunque los métodos iterativos de punto fijo convergen lentamente para muchos sistemas de ecuaciones, tienen propiedades que los hacen especialmente útiles para los métodos multimalla. En concreto, los métodos de punto fijo reducen eficazmente las componentes del vector de error en la dirección de los vectores propios (de la matriz de iteración del método iterativo de punto fijo) correspondientes a los valores propios de mayor magnitud. Estos vectores propios se denominan modos oscilatorios o de alta frecuencia ya que, para matrices que surgen en la discretización de EDPs elípticas, el valor de sus componentes oscila como una función periódica de alta frecuencia. El resto de componentes del vector de error, esto es, los asociados con modos suaves o de baja frecuencia, se reducen a un ritmo muy lento, con factores de reducción por iteración que pueden ser cercanos a la unidad para los valores propios más pequeños. Sin embargo, debido a las características inherentes a estos problemas, gran parte de los modos suaves correspondientes a la malla fina se pueden mapear con los modos oscilatorios de la malla inmediatamente más gruesa y, por lo tanto, resolviendo un sistema de ecuaciones similar al original en esta última malla se pueden atenuar eficazmente componentes del error que se reducirían muy lentamente aplicando exclusivamente el método de punto fijo en la malla fina. El tercer bloque básico de los métodos multimalla son *los operadores de transferencia entre mallas*. Estos operadores matriciales son necesarios para formular el sistema original en las distintas mallas de la jerarquía. Los operadores que transfieren el sistema desde una malla fina a la malla inmediatamente más gruesa se denominan *operadores de restricción*, y los de una malla gruesa a la malla inmediatamente más fina, *operadores de prolongación*. Una descripción de estos operadores y de cómo se definen apropiadamente para varios problemas modelo, y distintos tipos de malla y discretización, puede encontrarse en [80, 138] y en sus referencias.

La velocidad de convergencia de los métodos multimalla depende fuertemente de que sus bloques básicos exploten de la forma apropiada las particularidades del problema que se desea resolver. Teniendo en cuenta que no siempre es posible, fácil o intuitivo explotar la información particular de cada problema, esta propiedad dificulta considerablemente el desarrollo de métodos tan eficientes como los considerados para las EDP modelo. Además, el núcleo (e implementación) de los métodos multimalla depende de la jerarquía de mallas asociada al problema y, por lo tanto, su rango de aplicabilidad está limitado. Dado que los métodos multimalla son muy eficientes

para los sistemas de ecuaciones de *gran dimensión* que surgen en multitud de aplicaciones gobernadas por EDPs, diversos trabajos se han centrado en el desarrollo de métodos más generales basados en principios *similares* a los métodos multimalla. Los métodos algebraicos multimalla (*Algebraic MultiGrid*, AMG), también denominados métodos multinivel, son métodos iterativos diseñados con el propósito de extender los principios básicos de los métodos multimalla a sistemas generales, aunque esta extensión no es inmediata [138]. Debido a que la jerarquía de mallas no está disponible (o no se explota para derivar el método), los métodos AMG manejan otro tipo abstracciones conocidas como niveles. Cada uno de estos niveles tiene asociado un identificador, en lugar de un tamaño de malla, y los bloques básicos del método, como los operadores de restricción y de prolongación, se definen *algebraicamente*, es decir, explotando exclusivamente información que se deriva a partir de A. Aunque los métodos AMG no pueden considerarse como métodos de propósito general, su aplicabilidad es mucho mayor que la de los métodos multimalla. Por ejemplo, los métodos AMG son muy eficientes para resolver sistemas de gran dimensión que surgen de la discretización de EDPs elípticas mediante *mallas no estructuradas*, obteniendo en muchos casos velocidades de convergencia independientes de h. Sin embargo, para aplicaciones más generales, el mérito relativo de los métodos AMG cambia, y su robustez y eficiencia pueden disminuir significativamente en relación a otro tipo de métodos [113].

El estado del arte reconoce abiertamente la necesidad de incorporar de alguna forma el paradigma multinivel a los métodos de Krylov con el objetivo de conseguir resolutores competitivos para sistemas de gran dimensión. En esta línea, se ha desarrollado recientemente una nueva familia de precondicionadores derivados a partir de *variantes multinivel de la factorización ILU*. Esta familia de precondicionadores trata de mejorar la escalabilidad de los resolutores basados en la generación de subespacios de Krylov imitando, en cierta forma, a los métodos AMG. El enfoque más representativo de este tipo de métodos es el resolutor ARMS [140] (*Algebraic Recursive Multilevel Solver*). A continuación, describimos la estructura general de los precondicionadores derivados a partir de factorizaciones ILU multinivel, centrándonos en los bloques básicos que los integran. La justificación correspondiente a la definición de estos bloques básicos y su relación con los métodos AMG depende en gran medida del método de factorización ILU multinivel concreto, y queda fuera del ámbito de esta tesis.

Las factorizaciones ILU multinivel incorporan un mecanismo para separar el conjunto de las incógnitas del sistema original en dos subconjuntos disjuntos: el subconjunto fino (*fine set*) y el grueso (*coarse set*). Este mecanismo se denomina estrategia de selección de malla gruesa (*coarse grid selection strategy*) y, al igual que en los métodos AMG, gran parte del éxito de este tipo de factorizaciones ILU como precondicionadores depende del enfoque escogido para el diseño de la misma. En la práctica se han empleado diversos heurísticos como, por ejemplo, los basados en conjuntos independientes de incógnitas (*independent sets*) o su generalización, grupos de incógnitas independientes (*group independent sets*); para más detalles, consultar [113, 140] y sus referencias. Para la siguiente discusión, asumimos que los subconjuntos fino y grueso tienen, respectivamente, k y $n-k$ incógnitas. Si las incógnitas del sistema que pertenecen al subconjunto grueso se numeran tras las del subconjunto fino, entonces se obtiene un sistema permutado

$$Ax = b \rightarrow P^T APP^T x = P^T b \rightarrow P^T AP\hat{x} = \hat{b}, \tag{2.41}$$

que se puede particionar de la siguiente forma

$$P^T AP = \begin{pmatrix} B & F \\ E & C \end{pmatrix}, \ \hat{x} = \begin{pmatrix} \hat{x}_B \\ \hat{x}_C \end{pmatrix}, \ \hat{b} = \begin{pmatrix} \hat{b}_B \\ \hat{b}_C \end{pmatrix}, \tag{2.42}$$

donde $P \in \mathbb{R}^{n\times n}$ es la matriz de permutación correspondiente a la numeración mencionada, $B \in \mathbb{R}^{k\times k}$ contiene los coeficientes correspondientes al acoplamiento de las incógnitas que pertenecen al subconjunto fino, F y E, los de las incógnitas del subconjunto fino y grueso, y grueso y fino, respectivamente, y $C \in \mathbb{R}^{n-k\times n-k}$ los coeficientes correspondientes al acoplamiento de las incógnitas que pertenecen al subconjunto grueso.

La solución del sistema (2.42) se puede obtener mediante el Algoritmo 2.13, que se deriva fácilmente a partir de la siguiente factorización por bloques de P^TAP

$$P^TAP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ EB^{-1} & I \end{pmatrix} \begin{pmatrix} B & 0 \\ 0 & S_C \end{pmatrix} \begin{pmatrix} I & B^{-1}F \\ 0 & I \end{pmatrix}, \tag{2.43}$$

donde $S_C = C - EB^{-1}F$ es el complemento de Schur de C en P^TAP. En la práctica, el Algoritmo 2.13 sólo es útil si la resolución *directa* de sistemas de ecuaciones lineales con B y S_C en los pasos 2, 4 y 5 se puede llevar a cabo eficientemente, es decir, con un consumo moderado de tiempo y memoria. En caso contrario, *la factorización por bloques (2.43) puede emplearse como precondicionador del sistema (2.42)*, aunque para ello, es necesario calcular aproximaciones de B y S_C que se puedan "invertir" económicamente. Los precondicionadores derivados a partir de factorizaciones ILU multinivel explotan esta idea calculando una factorización ILU del bloque B, es decir, aproximando B como

$$B = \tilde{L}_B\tilde{U}_B + R_B. \tag{2.44}$$

Para calcular esta factorización se puede emplear cualquiera de los métodos básicos revisados en el Apartado 2.1.1. (Por ejemplo, el resolutor ARMS utiliza la factorización ILUT(τ,p).) Posteriormente, los factores aproximados $\tilde{L}_B$ y $\tilde{U}_B$ se emplean para "invertir" B^{-1} de forma aproximada en los pasos 2 y 5 del Algoritmo 2.13, así como para obtener la siguiente aproximación del complemento de Schur

$$S_C \approx \tilde{S}_C = C - (E\tilde{U}_B^{-1})(\tilde{L}_B^{-1}F). \tag{2.45}$$

La aplicación del precondicionador esbozado sobre un vector arbitrario $r \in \mathbb{R}^n$, para obtener el vector $z \in \mathbb{R}^n$, se muestra en el Algoritmo 2.14. Este algoritmo se deriva fácilmente teniendo en cuenta que, si las aproximaciones de B y S_C se obtienen conforme a (2.44) y (2.45), respectivamente, se obtiene la siguiente factorización (incompleta) por bloques de P^TAP

$$P^TAP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ E\tilde{U}_B^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \tilde{S}_C \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{L}_B^{-1}F \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & 0 \\ 0 & 0 \end{pmatrix}. \tag{2.46}$$

Algoritmo 2.13: Método directo de resolución de $Ax = b$ derivado a partir de (2.43).

1 $\begin{pmatrix} \hat{b}_B \\ \hat{b}_C \end{pmatrix} \leftarrow P^Tb$
2 $y_B \leftarrow B^{-1}\hat{b}_B$
3 $y_C \leftarrow \hat{b}_C + Ey_C$
4 Resolver $S_C\hat{x}_C = y_C$
5 $\hat{x}_B \leftarrow \hat{x}_B + B^{-1}F\hat{x}_C$
6 $x \leftarrow P\begin{pmatrix} \hat{x}_B \\ \hat{x}_C \end{pmatrix}$

Algoritmo 2.14: Aplicación del precondicionador derivado a partir de (2.46) a r.

1 $\begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} \leftarrow P^Tr$
2 $s_B \leftarrow \tilde{L}_B^{-1}\hat{r}_B$
3 $s_C \leftarrow \hat{r}_C - E\tilde{U}_B^{-1}s_B$
4 Resolver $\tilde{S}_C\hat{z}_C = s_C$
5 $\hat{z}_B \leftarrow s_B - \tilde{L}_B^{-1}F\hat{z}_C$
6 $\hat{z}_B \leftarrow \tilde{U}_B^{-1}\hat{z}_B$
7 $z \leftarrow P\begin{pmatrix} \hat{z}_B \\ \hat{z}_C \end{pmatrix}$

La implementación eficiente del algoritmo encargado de calcular la factorización (2.46) aproxima las matrices $E\tilde{U}_B^{-1}$ y $\tilde{L}_B^{-1}F$, extendiendo la factorización ILU (2.44) a los bloques E y F. Esto se puede llevar a cabo aplicando k pasos de la factorización ILUC (ver Algoritmo 2.7). Posteriormente, el complemento de Schur aproximado se calcula dentro de la rutina correspondiente a la factorización ILUC, aplicando al bloque C las actualizaciones retardadas tras las primeras k iteraciones. Como resultado de este proceso, se obtiene la siguiente factorización incompleta por bloques de P^TAP

$$P^TAP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & R_F \\ R_E & 0 \end{pmatrix}, \tag{2.47}$$

con

$$\tilde{L} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_E & I \end{pmatrix}, \ \tilde{U} = \begin{pmatrix} \tilde{U}_B & \tilde{U}_F \\ 0 & I \end{pmatrix}, \ R = \begin{pmatrix} R_B & R_F \\ R_E & 0 \end{pmatrix}. \tag{2.48}$$

El Algoritmo 2.7 descarta los elementos de $\tilde{L}$ y de $\tilde{U}$ situados en las posiciones correspondientes a los bloques no diagonales y, por lo tanto, $\tilde{L}_E = E\tilde{U}_B^{-1} + R_E$ y $\tilde{U}_F = \tilde{L}_B^{-1}F + R_F$, donde R_E y R_F contienen elementos "pequeños" descartados durante el proceso. Además, el complemento de Schur aproximado se define ahora como

$$S_C \approx \hat{S}_C = C - \tilde{L}_E\tilde{U}_F, \tag{2.49}$$

ya que se ha obtenido como resultado de aplicar al bloque C las actualizaciones retardadas tras las primeras k iteraciones.

El Algoritmo 2.14 no necesita los bloques $\tilde{L}_E$ y $\tilde{U}_F$ para aplicar el precondicionador derivado de la descomposición (2.47), y por tanto es posible ahorrar memoria *descartando estos bloques una vez que se han empleado para calcular* $\hat{S}_C$. Además, esta consideración permite que la acción del Algoritmo 2.14 sobre un vector arbitrario r se corresponda con la siguiente factorización por bloques:

$$P^TAP = \begin{pmatrix} B & F \\ E & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ E\tilde{U}_B^{-1} & I \end{pmatrix} \begin{pmatrix} I & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{U}_B & \tilde{L}_B^{-1}F \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & 0 \\ 0 & R_C \end{pmatrix}, \tag{2.50}$$

donde los errores causados al descartar elementos "pequeños" de los bloques no diagonales de $\tilde{L}$ y $\tilde{U}$ durante el proceso de aproximación, es decir, R_E y R_F, *han desaparecido de los bloques no diagonales de la matriz de error* R. Sin embargo, esta factorización sigue siendo menos precisa que la factorización (2.46) ya que, como consecuencia de calcular el completo de Schur conforme a (2.49), en lugar de a (2.45), surge un nuevo bloque en la matriz de error, que se define como $R_C = R_ER_F + E\tilde{U}_B^{-1}R_F + R_E\tilde{L}_B^{-1}F$. Para terminar este apunte de implementación, indicar que este enfoque se utiliza de forma invariable en los paquetes de software para el cálculo de factorización ILU multinivel (como ILUPACK o ARMS) con el objetivo de ahorrar memoria.

Un método iterativo precondicionado mediante la aplicación del Algoritmo 2.14 puede escalar pobremente con el tamaño de problema, ya que el sistema complemento de Schur sigue resolviéndose de forma directa en el paso 4. Para derivar métodos escalables, es necesario aproximar el complemento Schur. Las factorizaciones ILU multinivel construyen esta aproximación aplicando *recursivamente* a $\hat{S}_C$ el proceso descrito anteriormente (es decir, aplicando la estrategia de selección de malla gruesa y calculando la factorización incompleta por bloques). La construcción del precondicionador multinivel finaliza cuando, en un determinado nivel, la estrategia de selección de malla gruesa devuelve un subconjunto vacío, y el método aproxima o resuelve el sistema complemento de Schur. Para esta última decisión, se pueden utilizar distintos heurísticos basados, por ejemplo, en el tamaño o en la densidad de $\hat{S}_C$. Asimismo, se pueden usar diversos criterios heurísticos para detener la recursividad antes de que la estrategia de selección de malla gruesa devuelva un subconjunto vacío, como por ejemplo, que $\hat{S}_C$ sea lo suficientemente pequeño, que el tamaño de B sea relativamente pequeño o grande con respecto al de $\hat{S}_C$, o que se exceda un número máximo de niveles preestablecido.

2.2. Técnicas de precondicionado en ILUPACK

ILUPACK es una biblioteca numérica de precondicionado que implementa métodos para la resolución iterativa de sistemas de ecuaciones lineales dispersos y de gran dimensión. Las técnicas de precondicionado de la biblioteca se basan en una variante multinivel de la factorización ILU que se obtiene a partir de A de forma puramente algebraica, es decir, sin explotar la información específica de la aplicación que da lugar al sistema (1.1). Esta factorización se aplica como precondicionador del sistema (1.1), y el sistema precondicionado se resuelve empleando alguno de los métodos basados en la generación de subespacios de Krylov. La biblioteca proporciona técnicas de precondicionado y métodos iterativos específicos para cada tipo de sistema, considerando los casos SDP, simétrico indefinido, y no simétrico. Sin embargo, esta sección describe únicamente las técnicas de precondicionado específicas que incluye la biblioteca para el caso SDP, ya que la tesis se centra en la paralelización de estas técnicas.

La estructura de esta sección es la siguiente. En el Apartado 2.2.1 introducimos los fundamentos y objetivos de las técnicas de precondicionado basadas en la factorización ILU multinivel que incluye ILUPACK. En el Apartado 2.2.2 detallamos el núcleo básico de factorización ILU de la biblioteca y, en el Apartado 2.2.3, cómo se integra este núcleo con el resto de bloques básicos del método multinivel. Por último, en el Apartado 2.2.4 ilustramos el comportamiento del método cuando se aplica para calcular precondicionadores de los sistemas de ecuaciones que surgen en el contexto de una aplicación de ejemplo.

2.2.1. Fundamentos y objetivos

La descomposición LDL^T de una matriz $A \in \mathbb{R}^{n,n}$, SDP, se define como

$$A = \bar{L}\bar{L}^T = (LD^{1/2})(D^{1/2}L)^T = LDL^T, \tag{2.51}$$

donde $L \in \mathbb{R}^{n,n}$ es una matriz triangular inferior unidad y $D \in \mathbb{R}^{n,n}$ es una matriz diagonal con elementos positivos en la diagonal principal.

A su vez, la factorización incompleta LDL^T se define como

$$A = \tilde{L}\tilde{D}\tilde{L}^T + R, \tag{2.52}$$

donde $\tilde{L}$ y $\tilde{D}$ tienen la misma estructura que L y D, respectivamente, y R es la matriz de error que contiene aquellos elementos descartados durante el proceso de aproximación, típicamente aquellos que cumplen $|\tilde{l}_{ij}| \leq \tau$. Si aplicamos la factorización (2.52) como precondicionador del sistema original, conforme a (1.4), entonces obtenemos que la matriz del sistema precondicionado $(\tilde{L}\tilde{D}^{1/2})^{-1}A(\tilde{D}^{1/2}\tilde{L})^{-T}$ está relacionada con la matriz identidad a través de la siguiente expresión

$$(\tilde{L}\tilde{D}^{1/2})^{-1}A(\tilde{D}^{1/2}\tilde{L})^{-T} = I + \tilde{D}^{-1/2}\tilde{L}^{-1}R\tilde{L}^{-T}\tilde{D}^{-1/2} = I + \tilde{D}^{-1/2}Z\tilde{D}^{-1/2} = I + G, \tag{2.53}$$

donde $Z = \tilde{L}^{-1}R\tilde{L}^{-T}$ es la matriz de error inverso, y G es la matriz de perturbación que contiene las desviaciones del sistema precondicionado con respecto a la matriz identidad.

Los métodos basados en la generación de subespacios de Krylov convergen rápidamente si el sistema precondicionado (2.53) es una buena aproximación a la matriz identidad. Las técnicas de precondicionado de ILUPACK explotan esta propiedad controlando *directamente* la magnitud de las entradas de la matriz de error inverso, es decir, *controlando la estabilidad en la aplicación del factor aproximado* $\tilde{L}$. Para ello, la estrategia de selección de malla gruesa escoge la matriz de permutación P en (2.42), de tal forma que, en la siguiente factorización incompleta por bloques

$$\begin{aligned} P^TAP &= \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} + \begin{pmatrix} R_B & R_F^T \\ R_F & 0 \end{pmatrix} \\ &= \tilde{L}\tilde{D}\tilde{L}^T + R, \end{aligned} \tag{2.54}$$

el factor $\tilde{L}$ cumple que $\|\tilde{L}^{-1}\| \leq \kappa$, con $\kappa \geq 1$ una cota superior preestablecida, que se usa como parámetro para la estrategia de separación de las incógnitas del sistema original en los subconjuntos fino y grueso. Si aplicamos esta factorización como precondicionador del sistema (2.42), de forma acorde a (2.53), obtenemos la siguiente expresión para el sistema precondicionado

$$\begin{aligned}(\tilde{L}\tilde{D}^{1/2})^{-1}P^TAP(\tilde{D}^{1/2}\tilde{L})^{-T} &= I + \tilde{D}^{-1/2}\tilde{L}^{-1}R\tilde{L}^{-T}\tilde{D}^{-1/2} \\ &= I + \tilde{D}^{-1/2}Z\tilde{D}^{-1/2} \\ &= I + \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix}^{-1/2} \begin{pmatrix} Z_B & Z_E \\ Z_F & Z_C \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix}^{-1/2} \end{aligned} \quad (2.55)$$

donde la magnitud de las entradas de la matriz de error inverso es, como máximo, de orden $\tau\kappa^2$. Esta garantía no existe para la factorización aproximada de un sólo nivel (2.52) y el sistema precondicionado (2.53).

La factorización por bloques (2.54) sólo ha eliminado, de forma aproximada, las incógnitas del sistema correspondientes al subconjunto fino. La eliminación de las incógnitas correspondientes al subconjunto grueso se puede llevar a cabo resolviendo el sistema complemento de Schur de forma directa, o aproximándolo mediante una factorización incompleta del tipo LDL^T. La primera alternativa no introduce más error en (2.55), aunque el método resultante puede escalar pobremente con el tamaño del problema. La segunda alternativa completa la factorización parcial (2.54), obteniendo finalmente

$$P^TAP = \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & \tilde{L}_C \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{D}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & \tilde{L}_C^T \end{pmatrix} + \begin{pmatrix} R_B & R_F^T \\ R_F & R_C \end{pmatrix}, \quad (2.56)$$

con

$$\hat{S}_C = \tilde{L}_C\tilde{D}_C\tilde{L}_C^T + R_C. \quad (2.57)$$

Aplicando esta factorización de dos niveles a (2.42), el sistema precondicionado queda finalmente como

$$\begin{aligned}&\begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{D}_C \end{pmatrix}^{-1/2} \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & \tilde{L}_C \end{pmatrix}^{-1} P^TAP \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & \tilde{L}_C \end{pmatrix}^{-T} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{D}_C \end{pmatrix}^{-1/2} \\ &\qquad = I + \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{D}_C \end{pmatrix}^{-1/2} W \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \tilde{D}_C \end{pmatrix}^{-1/2}, \end{aligned} \quad (2.58)$$

donde W, la matriz de error inverso correspondiente a la factorización (2.56), cumple que

$$\begin{aligned} W = {} & Z && (2.59) \\ & + \begin{pmatrix} 0 & 0 \\ 0 & (\tilde{L}_C^{-1} - I)\tilde{R}_C(\tilde{L}_C^{-T} - I) \end{pmatrix} && (2.60) \\ & + \begin{pmatrix} 0 & 0 \\ 0 & \tilde{L}_C^{-1} - I \end{pmatrix} R\tilde{L}^{-T} && (2.61) \\ & + \tilde{L}^{-1}R \begin{pmatrix} 0 & 0 \\ 0 & \tilde{L}_C^{-T} - I \end{pmatrix}. && (2.62) \end{aligned}$$

Por lo tanto, el proceso que obtiene la aproximación de $\hat{S}_C$ en el segundo nivel, no sólo introduce perturbaciones en el sistema precondicionado a través del sumando (2.60), como ocurría con el proceso correspondiente al primer nivel y la matriz Z, sino que también puede amplificar de forma considerable el error inverso correspondiente al primer nivel, a través de los sumandos (2.61) y (2.62), si no se controla la magnitud de las entradas de la matriz $\tilde{L}_C^{-1}$. Esta propiedad motiva aún más, si cabe, la aplicación recursiva de la estrategia de selección de malla gruesa a $\hat{S}_C$, para de esa forma construir una factorización aproximada de $\hat{S}_C$ en la que $\|\tilde{L}_C^{-1}\| \leq \kappa$.

Recientemente se ha demostrado la conexión que existe entre la estrategia de selección de malla gruesa que da lugar a la factorización (2.54) y los *métodos algebraicos multinivel*. Para presentar esta conexión, consideramos el caso en el que no se aplica descarte durante el proceso de factorización, es decir, $R = 0$ en (2.54). Denotemos como $0 \leq \lambda_1 \leq \lambda_2 \cdots \leq \lambda_i \leq \cdots \leq \lambda_n$ los valores propios de A y como $0 \leq \mu_1 \leq \mu_2 \cdots \leq \mu_i \leq \cdots \leq \mu_m$ los valores propios de $\hat{S}_C$ en (2.54). Si $\tilde{L}$ en (2.54) cumple que $\|\tilde{L}^{-1}\|_2 \leq \kappa$, *entonces los valores propios de* $\hat{S}_C$ *se encuentran situados en el rango de los valores más pequeños de* A, es decir,

$$\lambda_i \leqslant \mu_i \leqslant \frac{\kappa^2}{1 - \|\tilde{D}_B^{-1}\|_2 \kappa \lambda_i} \lambda_i \approx \kappa^2 \lambda_i,$$

para todo i tal que $\lambda_i \ll 1/(\|\tilde{D}_B^{-1}\|_2 \kappa)$ (consultar [31] para la demostración). En términos de EDPs, este resultado analítico revela que la estrategia de selección de malla gruesa de ILUPACK "descubre" los valores propios más pequeños del sistema original en el "sistema grueso" $\hat{S}_C$. Además, también justifica la elección de un valor pequeño para κ (como por ejemplo $\kappa = 3$). Por lo tanto, el objetivo de conseguir que $\|\tilde{L}^{-1}\|$ sea menor que una cota superior moderada se descubre como una estrategia algebraica de engrosamiento (*algebraic coarsening strategy*). Típicamente, el sistema inicial no satisface $\|\tilde{L}^{-1}\| \leqslant \kappa$, a no ser que cumpla la siguiente propiedad de dominancia diagonal estricta

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ii}^{-1} a_{ij}| \leqslant 1 - \frac{1}{\kappa},$$

para $i = 1, \ldots, n-m$ (consultar [31]). Esta propiedad del sistema no es común en los problemas de aplicación reales y, en consecuencia, es necesario utilizar estrategias de pivotamiento para controlar el crecimiento de $\|L^{-1}\|$ durante el proceso de factorización. Estas estrategias son significativamente distintas a las que se utilizan en otras factorizaciones ILU para aumentar la estabilidad numérica o evitar elementos pivote nulos. Las estrategias de pivotamiento que incorpora el núcleo de factorización de ILUPACK rechazan una gran parte de las incógnitas del sistema (por ejemplo, el 30 % aproximadamente para la EDP de Laplace) durante el proceso de factorización. El siguiente apartado detalla el núcleo básico de ILUPACK y las estrategias de pivotamiento que éste incorpora.

2.2.2. Factorización incompleta LDL^T con pivotamiento basado en la inversa

El núcleo computacional que calcula la factorización (2.54) incorpora una estrategia de pivotamiento que controla $\|\tilde{L}^{-1}\|$ separando las incógnitas en los subconjuntos fino y grueso de forma dinámica, es decir, durante el proceso de eliminación. Esta estrategia se conoce en la literatura con el nombre de estrategia de pivotamiento basada en la inversa (*inverse-based pivoting strategy*) [34, 31, 141]. A continuación describimos con detalle una iteración correspondiente a la operación de la misma.

Por simplicidad, asumimos que, tras la eliminación de las primeras $1 < j-1 < n$ incógnitas, el estado de la factorización incompleta LDL^T con pivotamiento basado en la inversa es el siguiente:

$$A = \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{l}_{21}^T & 1 & 0 \\ \tilde{L}_{31} & 0 & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{11} & 0 & 0 \\ 0 & \hat{s}_{22} & \hat{s}_{32}^T \\ 0 & \hat{s}_{32} & \hat{S}_{33} \end{pmatrix} \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{l}_{21}^T & 1 & 0 \\ \tilde{L}_{31} & 0 & I \end{pmatrix}^T + \begin{pmatrix} R_{11} & r_{21} & R_{31}^T \\ r_{21}^T & 0 & 0 \\ R_{31} & 0 & 0 \end{pmatrix}, \quad (2.63)$$

donde los bloques $\tilde{L}_{11} \in \mathbb{R}^{j-1 \times j-1}$, $\tilde{l}_{21}^T \in \mathbb{R}^{1 \times j-1}$, $\tilde{L}_{31} \in \mathbb{R}^{n-j \times j-1}$ y $\tilde{D}_{11} \in \mathbb{R}^{j-1 \times j-1}$ se refieren a la parte ya calculada de la factorización incompleta, y los bloques $\hat{s}_{22} \in \mathbb{R}$, $\hat{s}_{32} \in \mathbb{R}^{n-j \times 1}$ y $\hat{S}_{33} \in \mathbb{R}^{n-j \times n-j}$ al complemento de Schur aproximado. Al comienzo de la iteración j-ésima, la

estrategia de pivotamiento utiliza la norma de la inversa del factor resultante de eliminar una nueva columna, es decir,

$$\left\| \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{l}_{21}^T & 1 & 0 \\ \tilde{L}_{31} & \frac{\hat{s}_{32}}{\hat{s}_{22}} & I \end{pmatrix}^{-1} \right\| = \|\tilde{L}_j^{-1}\|, \tag{2.64}$$

para "aceptar" o "rechazar" el elemento pivote $\hat{s}_{22}$. En concreto, si se cumple que

$$\|\tilde{L}_j^{-1}\| \leq \kappa, \tag{2.65}$$

el método *acepta* $\hat{s}_{22}$, y la factorización procede de forma acorde al Algoritmo 2.8; es decir, se obtiene un nuevo elemento de $\tilde{D}$, $\tilde{d}_{22} = \hat{s}_{22}$, y una nueva columna de $\tilde{L}$, $\tilde{l}_{32}$, aplicando las reglas de descarte a $\hat{s}_{32}/\hat{s}_{22}$; y seguidamente, se actualiza el bloque restante del complemento de Schur aproximado, $\hat{S}_{33} \leftarrow \hat{S}_{33} - \tilde{l}_{23}\tilde{d}_{22}\tilde{l}_{23}^T$. Realmente, el método no obtiene $\|\tilde{L}_j^{-1}\|$ de forma exacta para comprobar (2.65), dado que este cálculo es costoso en relación al coste de la factorización incompleta y, por lo tanto, forzar (2.65) de forma estricta es inviable a nivel práctico. En su lugar, el método estima $\|\tilde{L}_j^{-1}\|$ paso a paso, según el Algoritmo 2.11. Al comienzo de la iteración j-ésima, la factorización invoca el paso j-ésimo del Algoritmo 2.11, empleando las entradas del bloque $\hat{s}_{32}/\hat{s}_{22}$ para calcular estimadores de $\|e_k^T\tilde{L}_j^{-1}\|_\infty$, para $k = j, \ldots, n$. Posteriormente, el estimador de mayor magnitud se emplea como substituto de $\|\tilde{L}_j^{-1}\|_\infty$ en (2.65), aceptando el elemento pivote $\hat{s}_{22}$ si $\|\tilde{L}_j^{-1}\|_\infty \lessapprox \kappa$. En caso contrario, el método *rechaza* el elemento pivote, "moviendo" la fila y columna que ocupa éste a la posición que ocupan, respectivamente, la última fila y columna del sistema; además, las filas y columnas con identificador en el conjunto $\{j+1, \ldots, n\}$ se desplazan una posición hacia arriba y hacia la izquierda, respectivamente. Matemáticamente, esta operación se corresponde con una permutación simétrica, P, que transforma la factorización parcial (2.63) del siguiente modo

$$P^T A P = \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{31} & I & 0 \\ \tilde{l}_{21}^T & 0 & 1 \end{pmatrix} \begin{pmatrix} \tilde{D}_{11} & 0 & 0 \\ 0 & \hat{S}_{33} & \hat{s}_{32} \\ 0 & \hat{s}_{32}^T & \hat{s}_{22} \end{pmatrix} \begin{pmatrix} \tilde{L}_{11} & 0 & 0 \\ \tilde{L}_{31} & I & 0 \\ \tilde{l}_{21}^T & 0 & 1 \end{pmatrix}^T + \begin{pmatrix} R_{11} & R_{31}^T & r_{21} \\ R_{31} & 0 & 0 \\ r_{21}^T & 0 & 0 \end{pmatrix}, \tag{2.66}$$

donde el elemento pivote $\hat{s}_{22}$ se ha *rechazado*, y falta por factorizar el bloque

$$\begin{pmatrix} \hat{S}_{33} \\ \hat{s}_{32}^T \end{pmatrix}. \tag{2.67}$$

La factorización aplica el mismo proceso a este bloque, hasta que finalmente el complemento de Schur sólo contiene elementos pivote rechazados; esto es, hasta que se obtiene la factorización aproximada por bloques (2.54).

La implementación eficiente de la factorización LDL^T con pivotamiento basado en la inversa emplea la variante de Crout de la factorización LU; ver el Algoritmo 2.7 y las Figuras 2.3 y 2.4. La iteración j-ésima aplica las actualizaciones retardadas (en las primeras $j-1$ iteraciones) a la primera columna del bloque formado por la intersección de las últimas $n-j+1$ filas y columnas de A. De esta forma se obtienen $\hat{s}_{22}$ y $\hat{s}_{32}$ en (2.63), y *se evita actualizar el bloque* $\hat{S}_{33}$ *en cada iteración*[3]. Además, el bloque $\hat{s}_{32}^T$ no se calcula explícitamente, ya que el código explota la simetría del proceso para reducir a la mitad el coste computacional y de almacenamiento. Si la estrategia de pivotamiento rechaza el elemento pivote $\hat{s}_{22}$, entonces $\hat{s}_{22}$ y las entradas del bloque $\hat{s}_{32}$ se desechan, y la permutación sólo se aplica sobre la matriz del sistema y el factor triangular. Cuando se aplica este proceso paso a paso se obtiene finalmente la factorización aproximada por bloques (2.54), aunque falta calcular el complemento de Schur aproximado, $\hat{S}_C$.

[3]Esta operación no se puede implementar sin consumir un espacio nuevo de memoria para el bloque actualizado, ya que no se puede predecir el patrón de elementos no nulos del complemento de Schur aproximado y, además, las estructuras de datos no están preparadas para acomodar llenado dinámicamente (al menos de forma eficiente).

Este bloque se obtiene explícitamente, columna a columna, aplicando sobre C las actualizaciones correspondientes a la eliminación de las columnas de los elementos pivote aceptados, es decir, $\hat{S}_C = C - \tilde{L}_F \tilde{D}_B \tilde{L}_F^T$. La Figura 2.6 representa de forma gráfica el proceso a través del cual la factorización LDL^T con estrategia de pivotamiento basada en la inversa acaba produciendo la factorización incompleta por bloques (2.54). Los bloques en gris se refieren a bloques "intactos", es decir, bloques de la matriz $P^T AP$; los bloques con patrón a rayas, a bloques de los factores incompletos; y por último, los bloques con patrón cruzado, a bloques "actualizados", esto es, que pertenecen al complemento de Schur aproximado. Como se puede observar en la figura, cuando el bloque intacto de la descomposición contiene exclusivamente elementos pivote rechazados, entonces el complemento de Schur aproximado, $\hat{S}_C$, se calcula explícitamente.

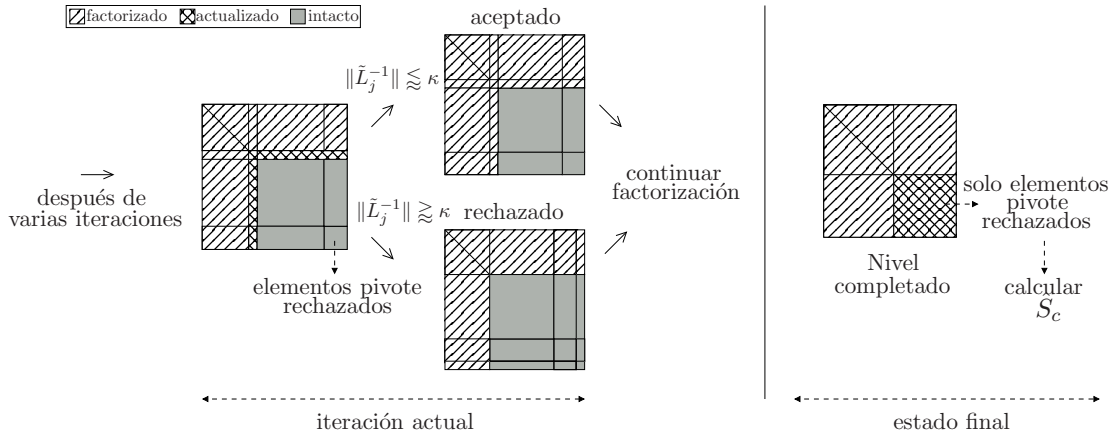


Figura 2.6: Factorización LDL^T con estrategia de pivotamiento basada en la inversa.

La factorización LDL^T de la biblioteca ILUPACK incorpora, por defecto, la técnica de Ajiz-Jennings [1] para descartar elementos de forma "segura". Para la siguiente explicación, asumimos que la factorización (2.63) acepta el elemento pivote $\hat{s}_{22}$ y, por lo tanto, para generar una nueva columna del factor aproximado, el método aplica las reglas de descarte al vector $\hat{s}_{32}/\hat{s}_{22}$. El complemento de Schur aproximado resultante

$$\begin{pmatrix} \hat{s}_{22} & \tilde{s}_{32}^T \\ \tilde{s}_{32} & \tilde{S}_{33} \end{pmatrix} = \begin{pmatrix} \hat{s}_{22} & \hat{s}_{32}^T \\ \hat{s}_{32} & \hat{S}_{33} \end{pmatrix} + \begin{pmatrix} 0 & \hat{r}_{32}^T \\ \hat{r}_{32} & 0 \end{pmatrix} \equiv \tilde{S} = \hat{S} + R, \tag{2.68}$$

con $\hat{r}_{32} = -\hat{s}_{32}$ en las posiciones correspondientes a elementos descartados y $\hat{r}_{32} = 0$ en el resto, puede ser *indefinido* potencialmente, causando problemas al resto de la factorización y la posterior resolución iterativa del sistema precondicionado (consultar el Apartado 2.1.2).

La técnica de Ajiz-Jennings genera una matriz error R en (2.68) simétrica semidefinida positiva, garantizando, en consecuencia, un complemento de Schur resultante, $\tilde{S}$, SDP [76]. Para describir la técnica, particionamos $\hat{S}^{(0)} = \hat{S}$ de la siguiente forma

$$\hat{S}^{(0)} = \left(\begin{array}{c|c} \hat{s}_{22}^{(0)} & (\hat{s}_{32}^{(0)})^T \\ \hline \hat{s}_{32}^{(0)} & \hat{S}_{33}^{(0)} \end{array}\right) \rightarrow \left(\begin{array}{c|cccccc} \hat{s}_{jj}^{(0)} & \hat{s}_{j+1j}^{(0)} & \cdots & \hat{s}_{kj}^{(0)} & \cdots & \hat{s}_{nj}^{(0)} \\ \hline \hat{s}_{j+1j}^{(0)} & \hat{s}_{j+1j+1}^{(0)} & \cdots & \hat{s}_{kj+1}^{(0)} & \cdots & \hat{s}_{nj+1}^{(0)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{s}_{kj}^{(0)} & \hat{s}_{kj+1}^{(0)} & \cdots & \hat{s}_{kk}^{(0)} & \cdots & \hat{s}_{nk}^{(0)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \hat{s}_{nj}^{(0)} & \hat{s}_{nj+1}^{(0)} & \cdots & \hat{s}_{nk}^{(0)} & \cdots & \hat{s}_{nn}^{(0)} \end{array}\right), \tag{2.69}$$

y asumimos que, al evaluar las reglas de descarte sobre los elementos del vector $\hat{s}_{32}/\hat{s}_{22}$, el método descarta los elementos $\hat{s}_{j+1j}/\hat{s}_{jj}$, $\hat{s}_{kj}/\hat{s}_{jj}$ y $\hat{s}_{nj}/\hat{s}_{nn}$, en este orden. En ILUPACK, un

elemento $\hat{s}_{mj}/\hat{s}_{jj}$ se descarta si se cumple que

$$\left|\frac{\hat{s}_{mj}}{\hat{s}_{jj}}\right| \max_{j\leq l\leq n}|\hat{s}_{lj}| \leq \frac{\tau}{\|e_j^T \tilde{L}_j^{-1}\|_\infty}|\hat{s}_{mm}|, \tag{2.70}$$

y

$$\left|\frac{\hat{s}_{mj}}{\hat{s}_{jj}}\right| \max_{j+1\leq l\leq n}\left|\frac{\hat{s}_{lj}}{\hat{s}_{ll}}\right| \leq \frac{\tau}{\|e_j^T \tilde{L}_j^{-1}\|_\infty}. \tag{2.71}$$

El superíndice en los términos de (2.69), como por ejemplo (0) en $\hat{S}^{(0)}$, indica el estado del elemento o matriz correspondiente después de haber descartado el número de elementos indicados por su valor.

Cuando se descarta el primer elemento, $\hat{s}_{j+1j}/\hat{s}_{jj}$, $\hat{S}^{(1)}$ se obtiene como

$$\hat{S}^{(1)} = \hat{S}^{(0)} + \left(\begin{array}{c|cccccc} r_{jj}^{(0)} & -\hat{s}_{j+1j}^{(0)} & \dots & 0 & \dots & 0 \\ \hline -\hat{s}_{j+1j}^{(0)} & r_{j+1j+1}^{(0)} & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 0 \end{array}\right) \equiv \hat{S}^{(1)} = \hat{S}^{(0)} + R^{(0)}, \tag{2.72}$$

donde la matriz $R^{(0)}$ es simétrica semidefinida positiva si se cumple que $r_{jj}^{(0)} > 0$, $r_{j+1j+1}^{(0)} > 0$ y $r_{jj}^{(0)} r_{j+1j+1}^{(0)} \geq (\hat{s}_{j+1j}^{(0)})^2$. Aunque existen infinitas soluciones para este sistema de inecuaciones, sólo interesan aquellos valores de $r_{jj}^{(0)}$, $r_{j+1j+1}^{(0)}$ con magnitud relativamente pequeña comparada con la de los elementos correspondientes de la diagonal principal de $\hat{S}^{(0)}$. Por ejemplo, los elementos de la diagonal principal de $R^{(0)}$ pueden definirse como $r_{jj}^{(0)} = r_{j+1j+1}^{(0)} = |\hat{s}_{j+1j}^{(0)}|$. La magnitud de $|\hat{s}_{j+1j}^{(0)}|$ es relativamente pequeña comparada con la de $\hat{s}_{jj}^{(0)}$, ya que de otra forma no se hubiese descartado el elemento $\hat{s}_{j+1j}/\hat{s}_{jj}$. Sin embargo, la magnitud de $|\hat{s}_{j+1j}^{(0)}|$ puede ser relativamente grande comparada con la de $\hat{s}_{j+1j+1}^{(0)}$, causando más error en este elemento de la diagonal principal de $\hat{S}^{(0)}$. En la técnica de Ajiz-Jennings los elementos de la diagonal principal de $R^{(0)}$ se definen como

$$r_{jj}^{(0)} = \sqrt{\frac{\hat{s}_{jj}^{(0)}}{\hat{s}_{j+1j+1}^{(0)}}}|\hat{s}_{j+1j}^{(0)}|, \; r_{j+1j+1}^{(0)} = \sqrt{\frac{\hat{s}_{j+1j+1}^{(0)}}{\hat{s}_{jj}^{(0)}}}|\hat{s}_{j+1j}^{(0)}|, \tag{2.73}$$

y, por tanto, la magnitud relativa de $r_{jj}^{(0)}$ con respecto a la de $\hat{s}_{jj}^{(0)}$ es equivalente a la de $r_{j+1j+1}^{(0)}$ con respecto a la de $\hat{s}_{j+1j+1}^{(0)}$. Además, se cumple que los valores propios de $\hat{S}^{(1)}$ no pueden ser más pequeños que los correspondientes en $\hat{S}^{(0)}$ y, por lo tanto, el condicionamiento del complemento de Schur aproximado no puede verse afectado negativamente por el efecto del descarte de elementos[4]; el Apartado 2.1.2 expone la problemática que podría causar el descarte en caso contrario. En cualquier caso, la robustez que proporciona esta técnica supone cierta pérdida de precisión.

[4]Esta última propiedad se cumple para cualquier matriz $R^{(0)}$ que sea simétrica semidefinida positiva [76].

Posteriormente, la técnica procede de forma acorde a (2.72) cuando se descarta $\hat{s}_{kj}/\hat{s}_{jj}$,

$$\hat{S}^{(2)} = \hat{S}^{(1)} + \left(\begin{array}{c|ccccc} \sqrt{\frac{\hat{s}_{jj}^{(1)}}{\hat{s}_{kk}^{(1)}}}|\hat{s}_{kj}^{(1)}| & 0 & \dots & -\hat{s}_{kj}^{(1)} & \dots & 0 \\ \hline 0 & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ -\hat{s}_{kj}^{(1)} & 0 & \dots & \sqrt{\frac{\hat{s}_{kk}^{(1)}}{\hat{s}_{jj}^{(1)}}}|\hat{s}_{kj}^{(1)}| & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 0 \end{array}\right) \equiv \hat{S}^{(2)} = \hat{S}^{(1)} + R^{(1)}, \qquad (2.74)$$

y $\hat{s}_{nj}/\hat{s}_{nn}$,

$$\hat{S}^{(3)} = \hat{S}^{(2)} + \left(\begin{array}{c|cccccc} \sqrt{\frac{\hat{s}_{jj}^{(2)}}{\hat{s}_{nn}^{(2)}}}|\hat{s}_{nj}^{(2)}| & 0 & \dots & 0 & \dots & -\hat{s}_{nj}^{(1)} \\ \hline 0 & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ -\hat{s}_{nj}^{(1)} & 0 & \dots & 0 & \dots & \sqrt{\frac{\hat{s}_{nn}^{(2)}}{\hat{s}_{jj}^{(2)}}}|\hat{s}_{nj}^{(2)}| \end{array}\right) \equiv \hat{S}^{(3)} = \hat{S}^{(2)} + R^{(2)}, \qquad (2.75)$$

obteniendo finalmente

$$\tilde{S} = \begin{pmatrix} \tilde{s}_{22} & \tilde{s}_{32}^T \\ \tilde{s}_{32} & \tilde{S}_{33} \end{pmatrix} = \hat{S} + R \equiv \hat{S}^{(3)} = \hat{S}^{(0)} + R, \qquad (2.76)$$

donde $R = R^{(0)} + R^{(1)} + R^{(2)}$ es una matriz simétrica semidefinida positiva; $\tilde{S}$, en consecuencia, es una matriz SDP; y $\tilde{s}_{j+1j} = \tilde{s}_{kj} = \tilde{s}_{nj} = 0$. La iteración j-ésima del proceso de factorización se completa a partir del complemento de Schur aproximado (2.76), obteniendo un nuevo elemento de $\tilde{D}$, $\tilde{d}_{22} = \tilde{s}_{22}$, una nueva columna de $\tilde{L}$, $\tilde{l}_{32} = \tilde{s}_{32}/\tilde{s}_{22}$ y, por último, actualizando el bloque restante del complemento de Schur aproximado, $\tilde{S}_{33} \leftarrow \tilde{S}_{33} - \tilde{l}_{23}\tilde{d}_{22}\tilde{l}_{23}^T$.

Las reglas de descarte (2.70) y (2.71), así como la técnica de Ajiz-Jennings, requieren acceso a los elementos de la diagonal principal del complemento de Schur aproximado. Esto acarrea ligeras modificaciones en las estructuras de datos que se emplean para almacenar las matrices involucradas en el proceso de factorización. En concreto, los elementos de la diagonal principal de $\tilde{D}$ y del complemento de Schur aproximado se almacenan en un vector denso. Al comienzo de la iteración j-ésima, este vector contiene los elementos $\tilde{d}_{11}, \dots, \tilde{d}_{j-1j-1}$ en las primeras $j-1$ posiciones, y los elementos $\hat{s}_{jj}, \dots, \hat{s}_{nn}$, en las $n-j+1$ restantes. Cuando se descartan los elementos $\hat{s}_{j+1j}/\hat{s}_{jj}$, $\hat{s}_{kj}/\hat{s}_{jj}$, y $\hat{s}_{nj}/\hat{s}_{nn}$, las posiciones j, $j+1$, k y n de este vector se actualizan conforme a (2.72), (2.74) y (2.75), obteniéndose los elementos $\tilde{s}_{jj}, \dots, \tilde{s}_{nn}$ de la diagonal principal de (2.76) en las últimas $n-j+1$ posiciones. Finalmente, las últimas $n-j$ posiciones se obtienen como la diagonal principal de la matriz $\tilde{S}_{33} \leftarrow \tilde{S}_{33} - \tilde{l}_{23}\tilde{d}_{22}\tilde{l}_{23}^T$.

2.2.3. Precondicionadores basados en la factorización LDL^T multinivel

Las técnicas de precondicionado de la biblioteca ILUPACK incorporan el núcleo de factorización LDL^T descrito en el apartado anterior como bloque básico para el cálculo del precondicionador multinivel. Para comenzar con la construcción del precondicionador, el método calcula la factorización aproximada por bloques (2.54) y, posteriormente, aplica recursivamente la misma estrategia al complemento de Schur aproximado, que constituye la matriz de entrada a procesar

en el siguiente nivel. Además, el método *preprocesa* la matriz de entrada a cada nivel con el objetivo de mejorar su condicionamiento y reducir el llenado que se produce durante el proceso de factorización. El proceso de construcción del precondicionador se puede expresar según el Algoritmo 2.15.

Algoritmo 2.15: Cálculo del precondicionador basado en la factorización LDL^T multinivel.

1. Calcular y aplicar matriz de escalado diagonal, $D = \text{diag}(a_{11}^{-1/2}, \ldots, a_{nn}^{-1/2})$, $A \to DAD = \tilde{A}$. Si A cumple ciertas propiedades [71], entonces $\kappa(A) \geq \kappa(\tilde{A})$.

2. Calcular y aplicar permutación para reducción de llenado $\tilde{P}$, $\tilde{A} \to \tilde{P}^T \tilde{A} \tilde{P} = \hat{A}$. Para este paso se puede utilizar cualquiera de los heurísticos de reducción de llenado mencionados en el Apartado 2.1.3, como por ejemplo, AMD, MLND, o RCM.

3. Calcular la factorización aproximada por bloques

$$P^T \hat{A} P = \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} + R, \tag{2.77}$$

   utilizando el núcleo de factorización LDL^T descrito en el Apartado 2.2.2. Una vez que se obtiene $\hat{S}_C$, el bloque $\tilde{L}_F$ se descarta para ahorrar memoria, y se guarda, en su lugar, el bloque F. Por lo tanto, la factorización aproximada por bloques que se acaba aplicando como precondicionador es la recogida en (2.50).

4. Proceder con el siguiente nivel repitiendo los pasos 1, 2 y 3 con $A \equiv \hat{S}_C$, hasta que finalmente $\hat{S}_C$ desaparezca o sea lo suficientemente denso como para ser gestionado eficientemente mediante un resolutor directo para matrices densas. Si $nnz(\hat{S}_C)$ representa el número de elementos no nulos del complemento de Schur aproximado, y $n-k$ su dimensión, entonces esta última decisión se toma si $nnz(\hat{S}_C)/(n-k)^2 > \nu$, con $0 \leq \nu \leq 1$, un parámetro prescrito. Por defecto, ILUPACK utiliza $\nu = 1/3$.

Además de los elementos de $\tilde{L}$ que se descartan durante la factorización del paso 3, el Algoritmo 2.15 descarta elementos adicionales del complemento de Schur aproximado, $\hat{S}_C$, una vez que se ha obtenido la factorización aproximada por bloques (2.77); es decir, entre el paso 3 y el 4. Para descartar estos elementos, ILUPACK utiliza por defecto una tolerancia de descarte, $\hat{\tau}$, un orden de magnitud inferior a la tolerancia usada para las reglas (2.70) y (2.71); esto es, $\hat{\tau} = \tau/10$. Un elemento $\hat{s}_{ij}$ de $\hat{S}_C$, con $i \neq j$, se descarta si se cumple la regla

$$\frac{|\hat{s}_{ij}|}{min(\hat{s}_{ii}, \hat{s}_{jj})} \|e_k^T \tilde{L}^{-1}\|_\infty \leq \hat{\tau}. \tag{2.78}$$

Por cada elemento que cumple esta condición, se utiliza la técnica de Ajiz-Jennings para asegurar que el complemento de Schur aproximado resultante sea simétrico definido positivo.

En la parte superior de la Figura 2.7 se muestra el patrón de elementos no nulos correspondiente a la factorización multinivel LDL^T calculada por el Algoritmo 2.15, con una matriz dispersa de ejemplo, $\tau = 10^{-3}$, $\kappa = 5$, y AMD como heurístico de reducción de llenado para el paso 2. Los elementos no nulos de color verde y azul se corresponden con elementos de los factores aproximados del bloque B en (2.77), es decir, $\tilde{L}_B$, $\tilde{L}_B^T$, $\tilde{D}_B$, y los de color rojo con elementos de los bloques F y F^T. Como se puede observar en la figura, el precondicionador construido consta de cinco niveles. La parte inferior de la Figura 2.7 es una ampliación de la parte superior que muestra con más detalle el patrón correspondiente a los últimos cuatro niveles. En el último nivel, el método ha invocado a un resolutor para matrices densas con el objetivo de resolver de forma directa el sistema complemento de Schur.

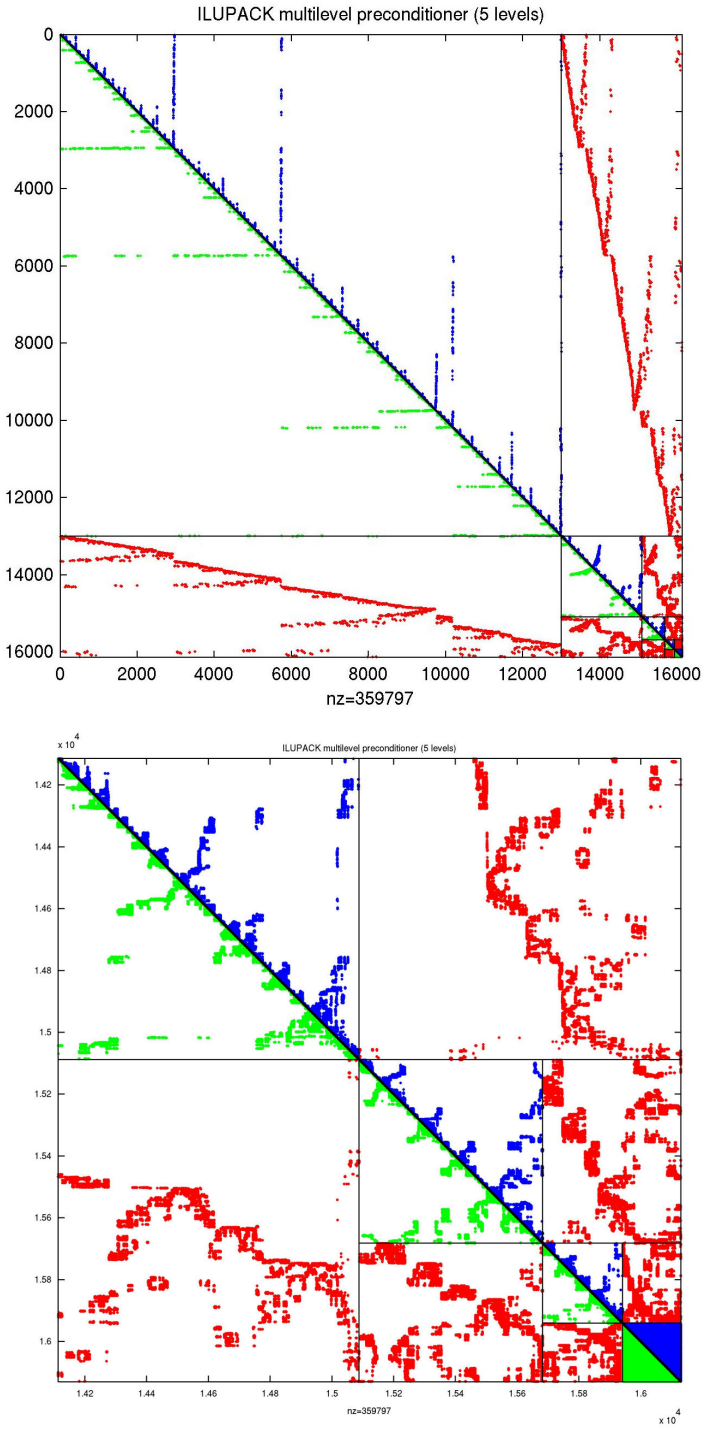


Figura 2.7: Patrón de elementos no nulos de la factorización multinivel LDL^T de una matriz dispersa de ejemplo. $\tau = 10^{-3}$, $\kappa = 5$, y AMD como heurístico de reducción de llenado.

2.2.4. Ejemplo de aplicación a la EDP de difusión en estado estacionario

En este apartado ilustramos brevemente el comportamiento de las técnicas de precondicionado de ILUPACK cuando se aplican a sistemas de ecuaciones que surgen de la discretización mediante el método de las diferencias finitas de dos variantes de la *EDP de difusión en estado estacionario*. La resolución numérica de EDPs se emplea comúnmente para evaluar el rendimiento de los métodos iterativos precondicionados, dada su importancia como herramienta para el modelado de fenómenos físicos. El objetivo del apartado es plasmar que el comportamiento del método de ILUPACK depende en gran medida de la aplicación de la que surge el sistema de ecuaciones que se pretende precondicionar, más que realizar un estudio exhaustivo y sistemático de su rendimiento; este tipo estudios pueden encontrarse en [34, 31, 141].

Los sistemas de ecuaciones que consideramos en este apartado surgen de la EDP de difusión bidimensional en estado estable

$$-\nabla \cdot \begin{bmatrix} a(x,y) & 0 \\ 0 & b(x,y) \end{bmatrix} \nabla u(x,y) = 0, \tag{2.79}$$

en el cuadrado unidad $\Omega = [0,1] \times [0,1]$, con condiciones de contorno de Dirichlet homogéneas; es decir, $u(x,0) = u(x,1) = u(0,y) = u(1,y) = 0$.

Para obtener soluciones aproximadas a la solución u de (2.79), sustituimos la región continua Ω, por una malla uniforme de $N \times N$ puntos $\Omega_h = \{(x_i = ih, y_j = jh) : i = 1,\ldots,N, j = 1,\ldots,N\}$, situados a distancia $h = 1/(N+1)$, y usamos el esquema estándar de diferencias finitas centradas para aproximar la derivada parcial en la dirección del eje x en un punto (x_i, y_j) dado de la malla, es decir,

$$\left(\frac{\partial}{\partial x} a \frac{\partial u}{\partial x}\right)(x_i, y_j) \approx \frac{a_{i+1/2,j}(u_{i+1,j} - u_{i,j}) - a_{i-1/2,j}(u_{i,j} - u_{i-1,j})}{h^2}, \tag{2.80}$$

donde $a_{i+1/2,j} = a(x_i + h/2, y_j)$, $a_{i-1/2,j} = a(x_i - h/2, y_j)$, y u_{ij} representa la aproximación de $u(x_i, y_j)$. Para aproximar la derivada parcial en la dirección del eje y, empleamos la expresión análoga

$$\left(\frac{\partial}{\partial x} b \frac{\partial u}{\partial x}\right)(x_i, y_j) \approx \frac{b_{i,j+1/2}(u_{i,j+1} - u_{i,j}) - b_{i,j-1/2}(u_{i,j} - u_{i,j-1})}{h^2}. \tag{2.81}$$

Sustituyendo (2.80) y (2.81) por sus homólogas continuas en (2.79), para $i = 1,\ldots,N$, $j = 1,\ldots,N$, obtenemos el siguiente sistema algebraico de $N \times N$ ecuaciones e incógnitas

$$\begin{aligned} -\frac{1}{h^2}\big((a_{i+1/2,j}(u_{i+1,j} - u_{i,j}) - a_{i-1/2,j}(u_{i,j} - u_{i-1,j}) + \\ b_{i,j+1/2}(u_{i,j+1} - u_{i,j}) - b_{i,j-1/2}(u_{i,j} - u_{i,j-1})\big) = 0. \end{aligned} \tag{2.82}$$

Con respecto a la elección de a y b, consideramos dos casos o variantes de (2.79): el caso *homogéneo*, es decir, $a(x,y) = b(x,y) = 1$, y el *anisotrópico*, $a(x,y) = 0,01$, $b(x,y) = 1$. Debido a que las computaciones numéricas se han realizado con MATLAB en un ordenador de sobremesa, consideraremos problemas de tamaño moderado, $N = 200$, aunque el comportamiento observado es representativo del que ofrecen problemas de mayor dimensión. Con respecto al ajuste del Algoritmo 2.15, hemos empleado la *ordenación natural* para la matriz de coeficientes del sistema (consultar el Apartado 2.1.3), desactivado el empleo de heurísticos de reducción de llenado (es decir, $P = I$ en el paso 2 del Algoritmo 2.15) y, por último, hemos escogido $\kappa = 3$ y $\tau = 10^{-3}$ para el núcleo de factorización con pivotamiento basado en la inversa (ver paso 3 del Algoritmo 2.15). La justificación *teórica y experimental* del porqué de esta elección del valor de κ es apropiada para este tipo de problemas puede encontrarse en [31].

La Figura 2.8 ilustra el número y distribución de las incógnitas (nodos en la malla) aceptadas (puntos azules) y rechazadas (puntos rojos) por la factorización LDL^T con pivotamiento basada en la inversa en los primeros cuatro niveles del proceso de construcción del precondicionador y, la

Figura 2.9, para los tres últimos. Como se puede observar en las figuras, el número y distribución de nodos aceptados y rechazados por el método en cada nivel es distinto para cada caso. En concreto, el método rechaza mayor número de nodos en el caso homogéneo (columna izquierda) que en el anisotrópico (columna derecha). Este factor influye sobre el coste computacional del método así como en el número de niveles requeridos para la construcción del precondicionador; en el caso homogéneo, la factorización se completa en 7 niveles, mientras que en el anisotrópico lo hace en 6.

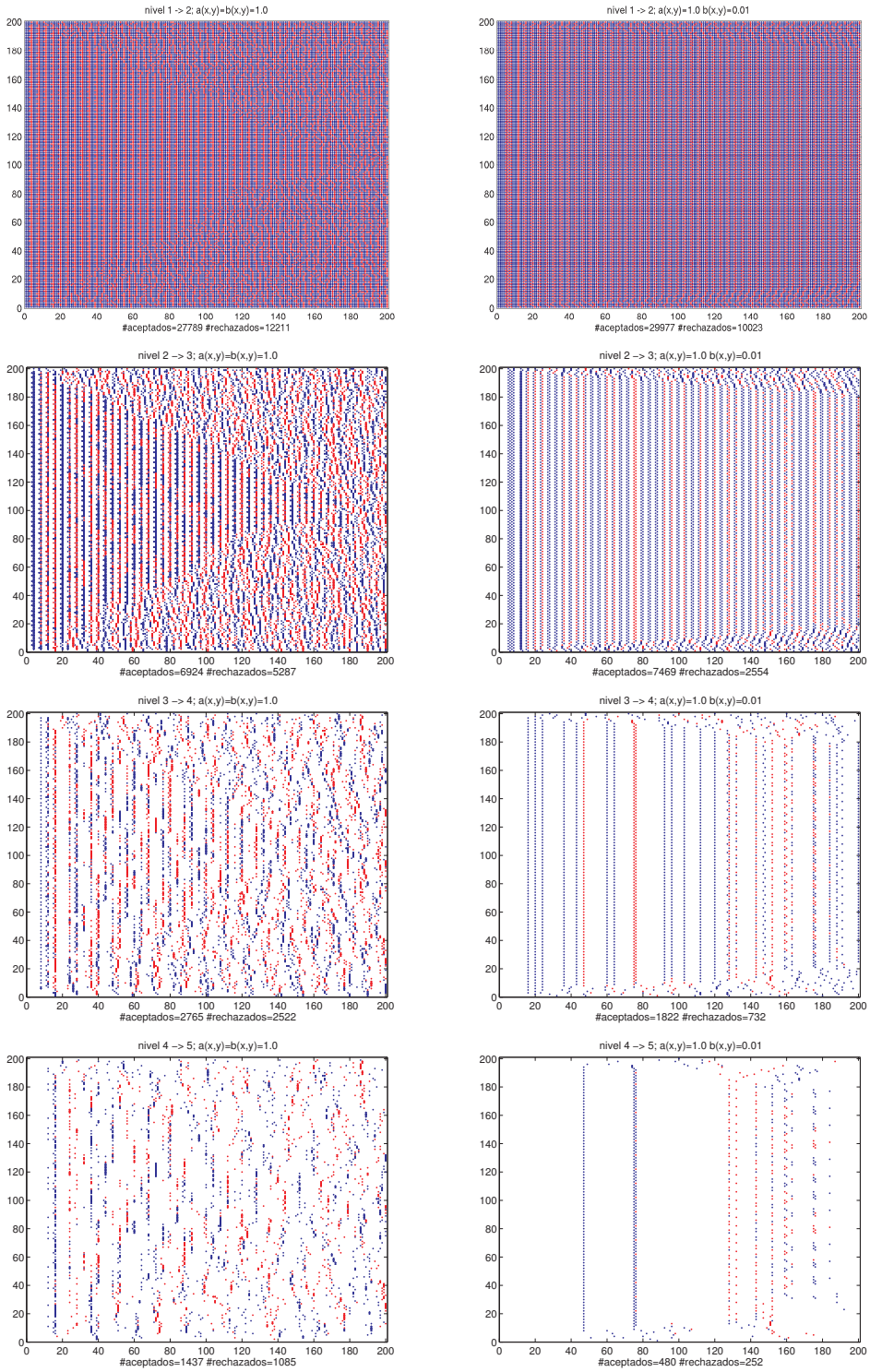


Figura 2.8: Número y distribución de las incógnitas aceptadas (puntos azules) y rechazadas (puntos rojos) en los primeros cuatro niveles del proceso de construcción del precondicionador. Izquierda: caso homogéneo. Derecha: caso anisotrópico.

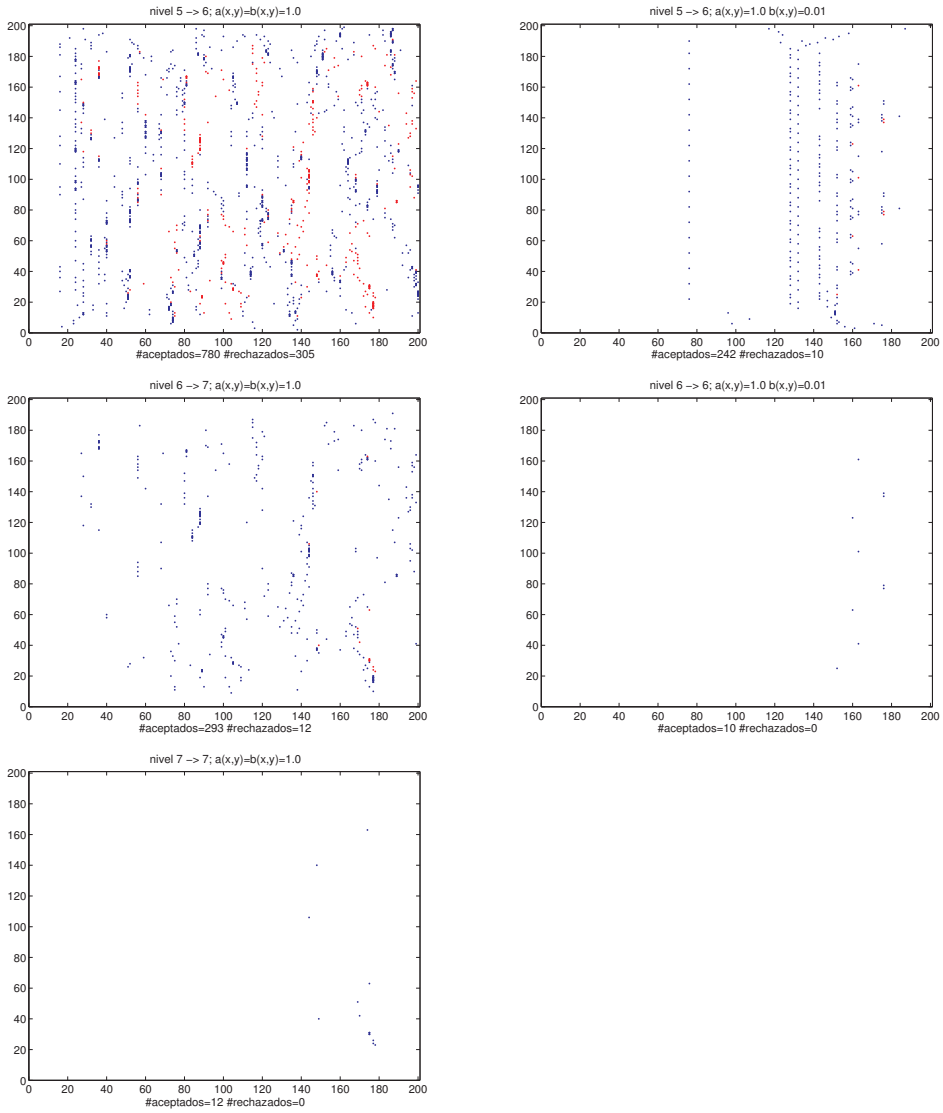


Figura 2.9: Número y distribución de las incógnitas aceptadas (puntos azules) y rechazadas (puntos rojos) en los últimos niveles del proceso de construcción del precondicionador. Izquierda: caso homogéneo. Derecha: caso anisotrópico.

2.3. Cálculo paralelo de precondicionadores LDL^T multinivel

En esta sección presentamos nuestro enfoque para la paralelización de las técnicas de precondicionado de ILÍACOS (Sección 2.2) para multiprocesadores con memoria compartida. Este enfoque combina dos ingredientes principales para identificar paralelismo de tareas en el cálculo de la factorización LDL^T multinivel. En primer lugar, se apoya sobre una fase inicial o preproceso que, particionando el grafo de adyacencia de la matriz del sistema, obtiene una reordenación de sus filas y columnas apropiada desde el punto de vista del paralelismo. En segundo lugar, el enfoque impone un cierto orden en el que se deben eliminar las incógnitas del sistema reordenado, transformando el método básico de ILUPACK para exponer un alto grado de paralelismo durante el cálculo de la factorización LDL^T multinivel.

La estructura de esta sección se describe a continuación. En el Apartado 2.3.1 nos centramos en el proceso que particiona el grafo de adyacencia de la matriz del sistema y la estructura de la reordenación a la que da lugar. El Apartado 2.3.2 describe detalladamente la transformación del algoritmo básico y analiza la concurrencia que se puede explotar en las computaciones que realiza este algoritmo. El Apartado 2.3.3 presenta cómo explota el algoritmo paralelo esta concurrencia a nivel de tareas a partir del árbol de dependencias; y el Apartado 2.3.4 completa la especificación del diseño de este algoritmo presentando las estrategias de mapeado y planificación de tareas a procesos. Por último, en el Apartado 2.3.5 ilustramos el rendimiento del método cuando se aplica para calcular precondicionadores paralelos de los sistemas de ecuaciones que surgen en el contexto de una aplicación de ejemplo.

2.3.1. División del problema: particionado del grafo de adyacencia

El problema de la disección de un grafo no dirigido se define de la siguiente forma: dado un grafo no dirigido, $G = \{V, E\}$, con V el conjunto de vértices y E el de aristas, obtener un subconjunto de vértices de cardinalidad mínima, $S \subseteq V$, denominado *subconjunto separador de vértices*, que divida G en dos subgrafos

$$G_1 = \{V_1, E_1\}, \quad G_2 = \{V_2, E_2\}, \tag{2.83}$$

con

$$V_1 \cup V_2 \cup S = V, \quad V_1 \cap S = \emptyset, \quad V_2 \cap S = \emptyset \tag{2.84}$$

y

$$E_1 = \{\{i,j\} \in E : i \in V_1 \wedge j \in V_1\} \quad \text{y} \quad E_2 = \{\{i,j\} \in E : i \in V_2 \wedge j \in V_2\}, \tag{2.85}$$

de modo que estos subgrafos sean de igual tamaño, es decir, $|V_1| = |V_2|$, e *independientes*

$$V_1 \cap V_2 = \emptyset \quad \text{y} \quad \nexists\, (i,j) \in E : i \in V_1 \wedge j \in V_2. \tag{2.86}$$

A pesar de que este problema es NP-completo, el estado del arte incluye heurísticos rápidos y eficientes [89, 102] para obtener soluciones aproximadas de calidad razonable, es decir, heurísticos que obtienen separadores de vértices de baja cardinalidad y subgrafos independientes con aproximadamente el mismo tamaño. De aquí en adelante, nos referiremos a estos heurísticos como *heurísticos para el cálculo de separadores de vértices*.

La disección anidada es una estrategia de *reordenación* para reducción de llenado que también resulta apropiada desde el punto de vista de la factorización paralela (directa o incompleta) de matrices dispersas simétricas [74, 89, 102]. Esta estrategia numera los vértices del grafo de adyacencia de la matriz del sistema mediante un proceso que aplica *recursivamente* heurísticos para el cálculo de separadores de vértices. El resultado es una reordenación de las filas y columnas de la matriz del sistema, $A \to \Pi^T A \Pi$, cuya estructura es apropiada como primer ingrediente para exponer paralelismo de tareas en el cálculo de la factorización LDL^T multinivel; Π se refiere a la matriz de permutación que la disección anidada obtiene como resultado del proceso

de numeración del grafo de adyacencia. A continuación describimos con más detalle la disección anidada, y la estructura de la reordenación a la que da lugar.

La disección anidada comienza dividiendo el grafo de adyacencia del sistema, $G(A)$, en dos subgrafos independientes, $G_{(2,1)}$ y $G_{(2,2)}$, a través de un separador de vértices, $S_{(1,1)}$. La matriz de permutación Π se calcula numerando, en primer lugar, los vértices de $G_{(2,1)}$ y $G_{(2,2)}$, y, a continuación, los del separador $S_{(1,1)}$. Por su parte, la numeración de $G_{(2,1)}$ y $G_{(2,2)}$ se obtiene aplicando *recursivamente* la misma idea, es decir, el proceso divide $G_{(2,1)}$ en los subgrafos $G_{(3,1)}$ y $G_{(3,2)}$, mediante un separador, $S_{(2,1)}$; y $G_{(2,2)}$, en $G_{(3,3)}$ y $G_{(3,4)}$, mediante $S_{(2,2)}$. Cuando los subgrafos independientes cumplen cierto criterio preestablecido, como por ejemplo que su número de vértices sea inferior a un umbral, entonces la recursividad finaliza, y los vértices de éstos se numeran utilizando heurísticos alternativos de ordenación para reducción de llenado, como por ejemplo, alguna variante del algoritmo del mínimo grado [14]. La Figura 2.10 muestra de forma gráfica cómo la disección anidada particiona el grafo de adyacencia para calcular una reordenación de la matriz del sistema. En la Figura 2.10 (c), el grafo de adyacencia correspondiente a la matriz del sistema, $G(A)$, ha sido particionado recursivamente en cuatro subgrafos independientes, $G_{(3,1)}$, $G_{(3,2)}$, $G_{(3,3)}$ y $G_{(3,4)}$, a través del separador $S_{(1,1)}$ en el primer nivel de la recursividad, y los separadores $S_{(2,1)}$ y $S_{(2,2)}$, en el segundo. En el tercer y último nivel, es decir, el correspondiente a los subgrafos independientes, la recursividad finaliza y los vértices de éstos se numeran utilizando heurísticos alternativos de ordenación para reducción de llenado. Sin embargo, en la Figura 2.10 (d), los vértices correspondientes a los subgrafos independientes se han numerado secuencialmente por filas, por cuestiones de simplicidad. La Figura 2.10 (e) no muestra el patrón de elementos no nulos de cada uno de los bloques de $\Pi^T A \Pi$, ya que la ordenación escogida localmente para los subgrafos independientes y separadores no es relevante para presentar la dinámica del algoritmo paralelo.

Además de la permutación resultante de la disección anidada, Π, podemos definir una permutación alternativa, Ω, que se obtiene agrupando los nodos del grafo de adyacencia en función del nivel de la disección anidada al que pertenecen y, posteriormente, numerando estos nodos de mayor a menor nivel. Para el caso de la Figura 2.10 (c), Ω se obtiene numerando, en primer lugar, los nodos correspondientes a los subgrafos independientes, a continuación, los separadores del segundo nivel y, por último, el separador raíz; ver Figura 2.11. De aquí en adelante utilizaremos la letra l para referirnos al número de niveles de la disección anidada. Así, $l = 3$ en el ejemplo de las Figuras 2.10 y 2.11.

2.3.2. Transformación del algoritmo básico y análisis de concurrencia

El segundo ingrediente para la identificación de paralelismo de tareas es una *transformación* del Algoritmo 2.15 que divide la construcción del precondicionador en varias fases, una por cada uno de los l niveles de recursividad de la disección anidada. Dependiendo de la fase en la que se encuentre el algoritmo, la estrategia de selección de malla gruesa *no puede escoger ciertas incógnitas del sistema como integrantes del subconjunto fino*. Estas incógnitas se escogen estratégicamente con el objetivo de que, en la estructura de la matriz resultante de aplicar la estrategia de selección de malla gruesa, es decir, $P^T \hat{A} P$ en (2.77), se pueda seguir explotando concurrencia para la construcción del sistema complemento de Schur aproximado, y en la factorización posterior de éste, requisito indispensable para la eficiencia del algoritmo paralelo. El presente apartado familiariza al lector con el algoritmo transformado y analiza la concurrencia que se puede explotar, presentando paulatinamente en detalle las computaciones que éste efectúa para el ejemplo presentado en las Figuras 2.10 y 2.11. El Apartado 2.3.3 especifica cómo el algoritmo paralelo explota esta concurrencia eficientemente.

Para el siguiente desarrollo, nos referiremos a los bloques de la matriz $\Omega^T A \Omega$ (ver Figu-

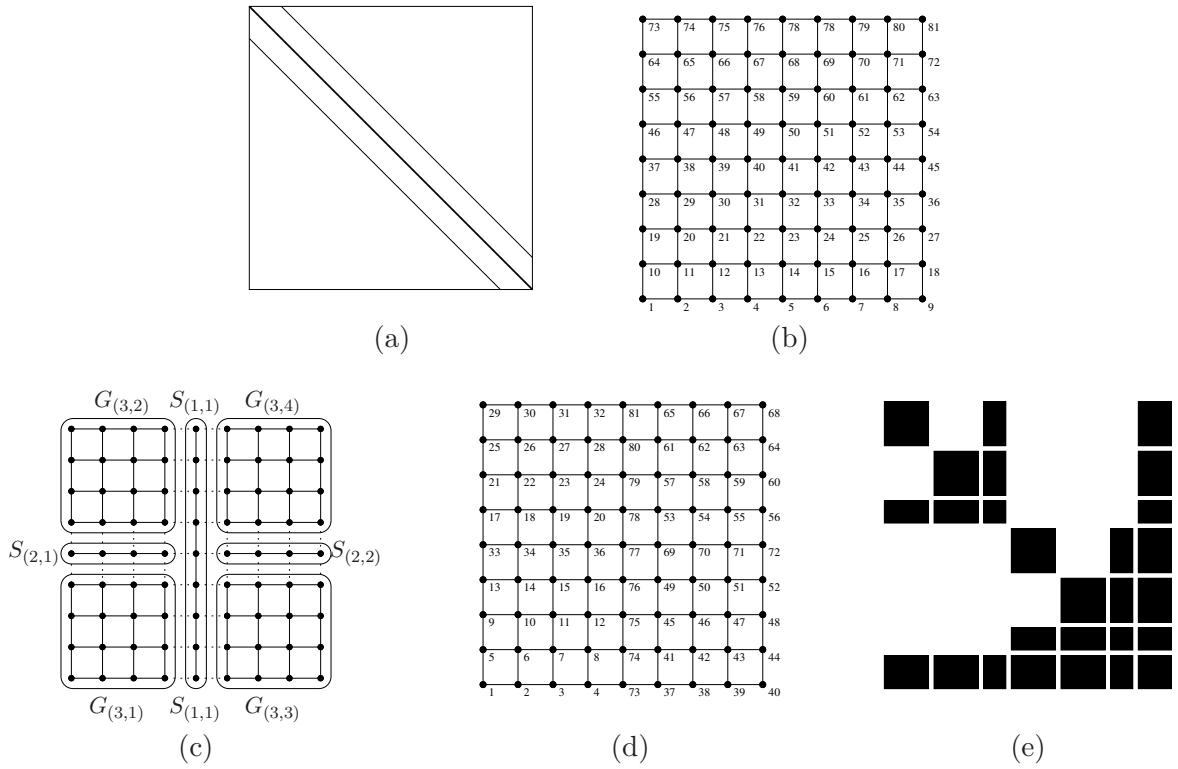


Figura 2.10: Conexión entre grafos no dirigidos y matrices dispersas simétricas, y el proceso de disección anidada. (a) Patrón de dispersidad de la ordenación natural de una matriz A. (b) Grafo de adyacencia asociado, $G(A)$, y numeración de los vértices correspondiente a la ordenación natural. (c) Disección anidada con tres niveles de recursividad. (d) Numeración de los vértices del grafo de adyacencia derivada de la disección anidada. (e) Estructura de la reordenación derivada de la disección anidada $A \rightarrow \Pi^T A \Pi$.

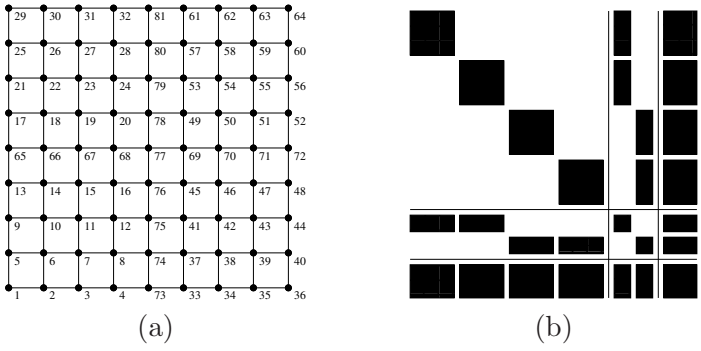


Figura 2.11: (a) Numeración por niveles de los nodos del grafo de adyacencia. (b) Reordenación derivada de la numeración por niveles $A \rightarrow \Omega^T A \Omega$.

ra 2.11 (b)) como

$$\Omega^T A\Omega = \left(\begin{array}{cccc|cc||c} A_{11}^{(3,1)} & 0 & 0 & 0 & (A_{51}^{(3,1)})^T & 0 & (A_{71}^{(3,1)})^T \\ 0 & A_{22}^{(3,2)} & 0 & 0 & (A_{52}^{(3,2)})^T & 0 & (A_{72}^{(3,2)})^T \\ 0 & 0 & A_{33}^{(3,3)} & 0 & 0 & (A_{63}^{(3,3)})^T & (A_{73}^{(3,3)})^T \\ 0 & 0 & 0 & A_{44}^{(3,4)} & 0 & (A_{64}^{(3,4)})^T & (A_{74}^{(3,4)})^T \\ \hline A_{51}^{(3,1)} & A_{52}^{(3,2)} & 0 & 0 & A_{55}^{(2,1)} & 0 & (A_{75}^{(2,1)})^T \\ 0 & 0 & A_{63}^{(3,3)} & A_{64}^{(3,4)} & 0 & A_{66}^{(2,2)} & (A_{76}^{(2,2)})^T \\ \hline\hline A_{71}^{(3,1)} & A_{72}^{(3,2)} & A_{73}^{(3,3)} & A_{74}^{(3,4)} & A_{75}^{(2,1)} & A_{76}^{(2,2)} & A_{77}^{(1,1)} \end{array}\right) \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right), \tag{2.87}$$

y emplearemos la siguiente notación

$$A \equiv \Omega^T A\Omega \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right),$$

para indicar, por una parte, que $\Omega^T A\Omega$ es la matriz de entrada, A, para la primera fase del algoritmo y, por otra, que esta matriz se puede reparticionar en una matriz 2×2 por bloques, donde A_X es diagonal por bloques.

Procesamiento de los subgrafos independientes

La primera fase del algoritmo procesa *únicamente* los nodos de los subgrafos independientes. Estos nodos se separan en los subconjuntos fino y grueso aplicando el mismo enfoque que el Algoritmo 2.15 y, durante esta fase, la estrategia selección de malla gruesa no puede seleccionar nodos de los separadores como integrantes del subconjunto fino; por lo tanto, los nodos de los separadores forman parte del subconjunto grueso. Para implementar esta estrategia sobre (2.87), la factorización LDL^T con pivotamiento basado en la inversa factoriza A_X y los bloques no diagonales correspondientes, A_Y e A_Y^T, de forma acorde al núcleo del Apartado 2.2.2, y no considera los elementos pivote correspondientes a A_Z. Como detallamos a continuación, la estructura de $A \equiv \Omega^T A\Omega$ permite extraer concurrencia en el cálculo de esta factorización.

En primer lugar, dado que A_X tiene estructura diagonal por bloques, cada uno de los bloques diagonales, $A_{ii}^{(3,i)}$ con $i = 1, \ldots, 4$, se puede factorizar de modo independiente conforme a

$$\begin{aligned} (P_{ii}^{(3,i)})^T A_{ii}^{(3,i)} P_{ii}^{(3,i)} &= \begin{pmatrix} B_{ii}^{(3,i)} & (F_{ii}^{(3,i)})^T \\ F_{ii}^{(3,i)} & C_{ii}^{(3,i)} \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{B,ii}^{(3,i)} & 0 \\ \tilde{L}_{F,ii}^{(3,i)} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B,ii}^{(3,i)} & 0 \\ 0 & \hat{S}_{C,ii}^{(3,i)} \end{pmatrix} \begin{pmatrix} (\tilde{L}_{B,ii}^{(3,i)})^T & (\tilde{L}_{F,ii}^{(3,i)})^T \\ 0 & I \end{pmatrix} + R_{ii}^{(3,i)}, \end{aligned} \tag{2.88}$$

donde $B_{ii}^{(3,i)}$ y $C_{ii}^{(3,i)}$ se corresponden, respectivamente, con los nodos aceptados y rechazados por la factorización LDL^T con pivotamiendo basado en la inversa de $A_{ii}^{(3,i)}$. De esta forma, se obtiene la siguiente factorización incompleta por bloques

$$\begin{aligned} P_X^T A_X P_X &= \begin{pmatrix} B_X & F_X^T \\ F_X & C_X \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{B_X} & 0 \\ \tilde{L}_{F_X} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B_X} & 0 \\ 0 & \hat{S}_{C_X} \end{pmatrix} \begin{pmatrix} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T \\ 0 & I \end{pmatrix} + R_X, \end{aligned} \tag{2.89}$$

donde los bloques B_X, F_X y C_X se pueden particionar como

$$B_X = \begin{pmatrix} B_{11}^{(3,1)} & 0 & 0 & 0 \\ 0 & B_{22}^{(3,2)} & 0 & 0 \\ 0 & 0 & B_{33}^{(3,3)} & 0 \\ 0 & 0 & 0 & B_{44}^{(3,4)} \end{pmatrix}, \; F_X = \begin{pmatrix} F_{11}^{(3,1)} & 0 & 0 & 0 \\ 0 & F_{22}^{(3,2)} & 0 & 0 \\ 0 & 0 & F_{33}^{(3,3)} & 0 \\ 0 & 0 & 0 & F_{44}^{(3,4)} \end{pmatrix},$$

$$C_X = \begin{pmatrix} C_{11}^{(3,1)} & 0 & 0 & 0 \\ 0 & C_{22}^{(3,2)} & 0 & 0 \\ 0 & 0 & C_{33}^{(3,3)} & 0 \\ 0 & 0 & 0 & C_{44}^{(3,4)} \end{pmatrix},$$

y $\tilde{L}_{B_X}$, $\tilde{L}_{F_X}$, $\tilde{D}_{B_X}$ y $\hat{S}_{C_X}$ de forma acorde a

$$\tilde{L}_{B_X} = \begin{pmatrix} \tilde{L}_{B,11}^{(3,1)} & 0 & 0 & 0 \\ 0 & \tilde{L}_{B,22}^{(3,2)} & 0 & 0 \\ 0 & 0 & \tilde{L}_{B,33}^{(3,3)} & 0 \\ 0 & 0 & 0 & \tilde{L}_{B,44}^{(3,4)} \end{pmatrix}, \; \tilde{L}_{F_X} = \begin{pmatrix} \tilde{L}_{F,11}^{(3,1)} & 0 & 0 & 0 \\ 0 & \tilde{L}_{F,22}^{(3,2)} & 0 & 0 \\ 0 & 0 & \tilde{L}_{F,33}^{(3,3)} & 0 \\ 0 & 0 & 0 & \tilde{L}_{F,44}^{(3,4)} \end{pmatrix},$$

$$\tilde{D}_{B_X} = \begin{pmatrix} \tilde{D}_{B,11}^{(3,1)} & 0 & 0 & 0 \\ 0 & \tilde{D}_{B,22}^{(3,2)} & 0 & 0 \\ 0 & 0 & \tilde{D}_{B,33}^{(3,3)} & 0 \\ 0 & 0 & 0 & \tilde{D}_{B,44}^{(3,4)} \end{pmatrix}, \; \hat{S}_{C_X} = \begin{pmatrix} \hat{S}_{C,11}^{(3,1)} & 0 & 0 & 0 \\ 0 & \hat{S}_{C,22}^{(3,2)} & 0 & 0 \\ 0 & 0 & \hat{S}_{C,33}^{(3,3)} & 0 \\ 0 & 0 & 0 & \hat{S}_{C,44}^{(3,4)} \end{pmatrix}.$$

En segundo lugar, si extendemos la factorización de A_X a A_Y, se puede observar que la factorización incompleta de cada uno de los bloques de A_Y se puede calcular independientemente a la del resto. Así, cada uno de los bloques $A_{7i}^{(7,i)}$, con $i = 1, \ldots, 4$, se puede factorizar de forma independiente como

$$\begin{aligned} A_{7i}^{(3,i)} P_{ii}^{(3,i)} &= \begin{pmatrix} F_{7i}^{(3,i)} & C_{7i}^{(3,i)} \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{F,7i}^{(3,i)} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B,ii}^{(3,i)} & 0 \\ 0 & \hat{S}_{C,7i}^{(3,i)} \end{pmatrix} \begin{pmatrix} (\tilde{L}_{B,ii}^{(3,i)})^T & (\tilde{L}_{F,ii}^{(3,i)})^T \\ 0 & I \end{pmatrix} + R_{7i}^{(3,i)}. \end{aligned} \tag{2.90}$$

Integrando la factorización incompleta de todos los bloques de A_Y, obtenemos

$$\begin{aligned} A_Y P_X &= \begin{pmatrix} F_Y & C_Y \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{F_Y} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B_X} & 0 \\ 0 & \hat{S}_{C_Y} \end{pmatrix} \begin{pmatrix} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T \\ 0 & I \end{pmatrix} + R_Y, \end{aligned} \tag{2.91}$$

donde F_Y y C_Y se definen como

$$F_Y = \left(\begin{array}{cccc} F_{51}^{(3,1)} & F_{52}^{(3,2)} & 0 & 0 \\ 0 & 0 & F_{63}^{(3,3)} & F_{64}^{(3,4)} \\ \hline F_{71}^{(3,1)} & F_{72}^{(3,2)} & F_{73}^{(3,3)} & F_{74}^{(3,4)} \end{array}\right), \; C_Y = \left(\begin{array}{cccc} C_{51}^{(3,1)} & C_{52}^{(3,2)} & 0 & 0 \\ 0 & 0 & C_{63}^{(3,3)} & C_{64}^{(3,4)} \\ \hline C_{71}^{(3,1)} & C_{72}^{(3,2)} & C_{73}^{(3,3)} & C_{74}^{(3,4)} \end{array}\right),$$

y $\tilde{L}_{F_Y}$ y $\hat{S}_{C_Y}$ como

$$\tilde{L}_{F_Y} = \left(\begin{array}{cccc} \tilde{L}_{F,51}^{(3,1)} & \tilde{L}_{F,52}^{(3,2)} & 0 & 0 \\ 0 & 0 & \tilde{L}_{F,63}^{(3,3)} & \tilde{L}_{F,64}^{(3,4)} \\ \hline \tilde{L}_{F,71}^{(3,1)} & \tilde{L}_{F,72}^{(3,2)} & \tilde{L}_{F,73}^{(3,3)} & \tilde{L}_{F,74}^{(3,4)} \end{array}\right), \; \hat{S}_{C_Y} = \left(\begin{array}{cccc} \hat{S}_{C,51}^{(3,1)} & \hat{S}_{C,52}^{(3,2)} & 0 & 0 \\ 0 & 0 & \hat{S}_{C,63}^{(3,3)} & \hat{S}_{C,64}^{(3,4)} \\ \hline \hat{S}_{C,71}^{(3,1)} & \hat{S}_{C,72}^{(3,2)} & \hat{S}_{C,73}^{(3,3)} & \hat{S}_{C,74}^{(3,4)} \end{array}\right).$$

Por último, si extendemos la construcción del complemento de Schur aproximado al bloque

A_Z, entonces

$$\hat{S}_C = \left(\begin{array}{cccc|cc||c} \hat{S}^{(3,1)}_{C,11} & 0 & 0 & 0 & (\hat{S}^{(3,1)}_{C,51})^T & 0 & (\hat{S}^{(3,1)}_{C,71})^T \\ 0 & \hat{S}^{(3,2)}_{C,22} & 0 & 0 & (\hat{S}^{(3,2)}_{C,52})^T & 0 & (\hat{S}^{(3,2)}_{C,72})^T \\ 0 & 0 & \hat{S}^{(3,3)}_{C,33} & 0 & 0 & (\hat{S}^{(3,3)}_{C,63})^T & (\hat{S}^{(3,3)}_{C,73})^T \\ 0 & 0 & 0 & \hat{S}^{(3,4)}_{C,44} & 0 & (\hat{S}^{(3,4)}_{C,64})^T & (\hat{S}^{(3,4)}_{C,74})^T \\ \hline \hat{S}^{(3,1)}_{C,51} & \hat{S}^{(3,2)}_{C,52} & 0 & 0 & \hat{S}^{(2,1)}_{C,55} & 0 & (\hat{S}^{(2,1)}_{C,75})^T \\ 0 & 0 & \hat{S}^{(3,3)}_{C,63} & \hat{S}^{(3,4)}_{C,64} & 0 & \hat{S}^{(2,2)}_{C,66} & (\hat{S}^{(2,2)}_{C,76})^T \\ \hline\hline \hat{S}^{(3,1)}_{C,71} & \hat{S}^{(3,2)}_{C,72} & \hat{S}^{(3,3)}_{C,73} & \hat{S}^{(3,4)}_{C,74} & \hat{S}^{(2,1)}_{C,75} & \hat{S}^{(2,2)}_{C,76} & \hat{S}^{(1,1)}_{C,77} \end{array}\right) \equiv \left(\begin{array}{c|c} \hat{S}_{C_X} & \hat{S}^T_{C_Y} \\ \hline \hat{S}_{C_Y} & \hat{S}_{C_Z} \end{array}\right), \tag{2.92}$$

e integrando esta expresión con (2.89) y (2.91), obtenemos la factorización parcial que se calcula en el primer nivel de la jerarquía

$$\left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right)^T \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right) \left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right) = \left(\begin{array}{cc|c} B_X & F_X^T & F_Y^T \\ F_X & C_X & C_Y^T \\ \hline F_Y & C_Y & C_Z \end{array}\right) = \tag{2.93}$$

$$\left(\begin{array}{cc|c} \tilde{L}_{B_X} & 0 & 0 \\ \tilde{L}_{F_X} & I & 0 \\ \hline \tilde{L}_{F_Y} & 0 & I \end{array}\right) \left(\begin{array}{cc|c} \tilde{D}_{B_X} & 0 & 0 \\ 0 & \hat{S}_{C_X} & \hat{S}^T_{C_Y} \\ \hline 0 & \hat{S}_{C_Y} & \hat{S}_{C_Z} \end{array}\right) \left(\begin{array}{cc|c} \tilde{L}^T_{B_X} & \tilde{L}^T_{F_X} & \tilde{L}^T_{F_Y} \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right) + \left(\begin{array}{c|c} R_X & R_Y^T \\ \hline R_Y & 0 \end{array}\right),$$

donde el bloque $\hat{S}_{C_Z}$ se construye a partir de los factores incompletos como

$$\begin{aligned} \hat{S}_{C_Z} &= C_Z - \tilde{L}_{F_Y}\tilde{D}_{B_X}\tilde{L}^T_{F_Y} \\ &= C_Z - \left(\begin{array}{cccc} \tilde{L}^{(3,1)}_{F,51} & \tilde{L}^{(3,2)}_{F,52} & 0 & 0 \\ 0 & 0 & \tilde{L}^{(3,3)}_{F,63} & \tilde{L}^{(3,4)}_{F,64} \\ \hline\hline \tilde{L}^{(3,1)}_{F,71} & \tilde{L}^{(3,2)}_{F,72} & \tilde{L}^{(3,3)}_{F,73} & \tilde{L}^{(3,4)}_{F,74} \end{array}\right) \tilde{D}_{B_X} \left(\begin{array}{cc||c} (\tilde{L}^{(3,1)}_{F,51})^T & 0 & (\tilde{L}^{(3,1)}_{F,71})^T \\ (\tilde{L}^{(3,2)}_{F,52})^T & 0 & (\tilde{L}^{(3,2)}_{F,72})^T \\ 0 & (\tilde{L}^{(3,3)}_{F,63})^T & (\tilde{L}^{(3,3)}_{F,73})^T \\ 0 & (\tilde{L}^{(3,4)}_{F,64})^T & (\tilde{L}^{(3,4)}_{F,74})^T \end{array}\right). \end{aligned}$$

Desarrollando esta última expresión, el substraendo se puede descomponer en los siguientes sumandos

$$\begin{aligned} \hat{S}_{C_Z} = C_Z - & \left[\left(\begin{array}{cc||c} \tilde{L}^{(3,1)}_{F,51}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,51})^T & 0 & \tilde{L}^{(3,1)}_{F,51}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F71})^T \\ 0 & 0 & 0 \\ \hline \tilde{L}^{(3,1)}_{F,71}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,51})^T & 0 & \tilde{L}^{(3,1)}_{F,71}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,71})^T \end{array}\right) + \right. \\ & \left(\begin{array}{cc||c} \tilde{L}^{(3,2)}_{F,52}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,52})^T & 0 & \tilde{L}^{(3,2)}_{F,52}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,72})^T \\ 0 & 0 & 0 \\ \hline \tilde{L}^{(3,2)}_{F,72}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,52})^T & 0 & \tilde{L}^{(3,2)}_{F,72}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,72})^T \end{array}\right) + \\ & \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & \tilde{L}^{(3,3)}_{F,63}\tilde{D}^{(3,3)}_{B,33}(\tilde{L}^{(3,3)}_{F,63})^T & \tilde{L}^{(3,3)}_{F,63}\tilde{D}^{(3,3)}_{B,33}(\tilde{L}^{(3,3)}_{F,73})^T \\ \hline 0 & \tilde{L}^{(3,3)}_{F,73}\tilde{D}^{(3,3)}_{B,33}(\tilde{L}^{(3,3)}_{F,63})^T & \tilde{L}^{(3,3)}_{F,73}\tilde{D}^{(3,3)}_{B,33}(\tilde{L}^{(3,3)}_{F,73})^T \end{array}\right) + \\ & \left.\left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & \tilde{L}^{(3,4)}_{F,64}\tilde{D}^{(3,4)}_{B,44}(\tilde{L}^{(3,4)}_{F,64})^T & \tilde{L}^{(3,4)}_{F,64}\tilde{D}^{(3,4)}_{B,44}(\tilde{L}^{(3,4)}_{F,74})^T \\ \hline 0 & \tilde{L}^{(3,4)}_{F,74}\tilde{D}^{(3,4)}_{B,44}(\tilde{L}^{(3,4)}_{F,64})^T & \tilde{L}^{(3,4)}_{F,74}\tilde{D}^{(3,4)}_{B,44}(\tilde{L}^{(3,4)}_{F,74})^T \end{array}\right)\right], \end{aligned} \tag{2.94}$$

donde el cálculo de cualquiera de estos sumandos se puede llevar a cabo de forma independiente a la del resto, aunque la acumulación de estos sumandos a $\hat{S}_{C_Z}$ requiere una operación de reduc-

ción (sincronización). Como veremos en el Apartado 2.3.3, el algoritmo paralelo gestiona esta acumulación de forma estratégica para aumentar de forma significativa el grado de concurrencia.

La Figura 2.12 captura la dinámica de un núcleo de factorización LDL^T con pivotamiento basado en la inversa que explota la concurrencia que se pone de manifiesto en la factorización de A_X, y los bloques no diagonales correspondientes, A_Y e A_Y^T, para calcular la factorización incompleta por bloques (2.93). Como se puede observar en la figura, los bloques de A_X, A_Y e A_Y^T, pueden ser procesados de modo independiente. Resulta necesario mencionar que, cuando se rechaza un elemento pivote de un bloque de A_X, la fila y columna correspondientes se "mueven", respectivamente, a la posiciones que ocupan la última fila y columna del bloque, a diferencia del núcleo de factorización del Algoritmo 2.15, que lo hace a las posiciones de la última fila y columna de la matriz completa.

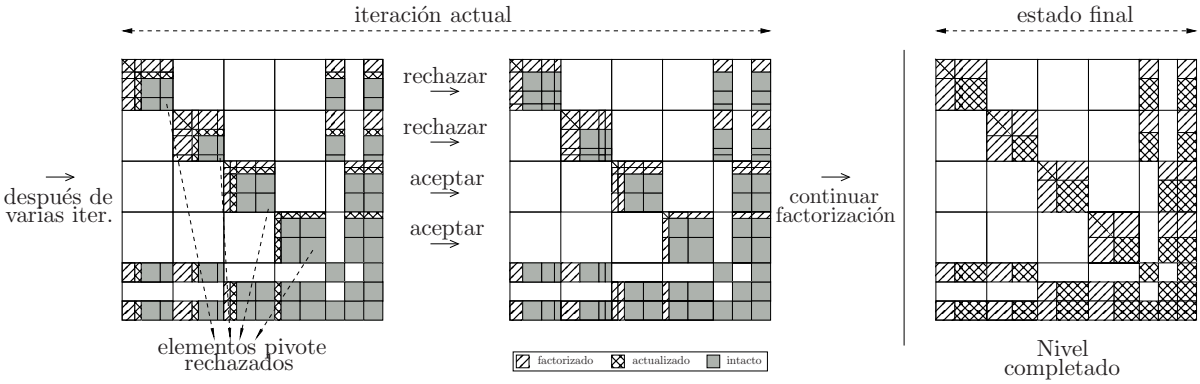


Figura 2.12: Factorización LDL^T con pivotamiento basado en la inversa que procesa de modo independiente A_X, y los bloques no diagonales correspondientes, A_Y e A_Y^T, para calcular la factorización incompleta por bloques (2.93).

La primera fase del algoritmo procede con el siguiente nivel, repitiendo la factorización incompleta por bloques (2.93) con $A \equiv \hat{S}_C$. La estructura del complemento de Schur aproximado (2.92) es equivalente a la de (2.87) y, por lo tanto, en el siguiente nivel, se puede seguir explotando concurrencia en el cálculo de (2.93). Esta relación entre la estructura de A y $\hat{S}_C$ es una consecuencia directa de las restricciones impuestas sobre la estrategia de selección de malla gruesa en esta fase del algoritmo.

Las computaciones detalladas hasta el momento para la primera fase se pueden expresar según el Algoritmo 2.16, que finaliza cuando el tamaño del bloque $\hat{S}_{C_X}$ en (2.93) es "suficientemente pequeño".

Algoritmo 2.16: Primera fase de la transformación del Algoritmo 2.15.

Entrada: $A \equiv \Omega^T A \Omega \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right)$

1. Calcular la factorización incompleta por bloques (2.93) del bloque A_X, utilizando el núcleo de factorización LDL^T descrito en el Apartado 2.2.2.

2. Proceder con el siguiente nivel repitiendo el paso 1 con

$$A \equiv \hat{S}_C \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right),$$

hasta que finalmente $\hat{S}_{C_X}$ en (2.93) quede vacío o sea lo "suficientemente pequeño".

A continuación, el algoritmo procede con una *nueva fase* cuya matriz de entrada, A, se

obtiene a partir de una reordenación de (2.92), $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$, con la siguiente estructura

$$
\begin{aligned}
A &\equiv \Phi^T \hat{S}_C \Phi \\
&= \left(\begin{array}{cccccc||c}
\hat{S}_{C,11}^{(3,1)} & 0 & (\hat{S}_{C,51}^{(3,1)})^T & 0 & 0 & 0 & (\hat{S}_{C,71}^{(3,1)})^T \\
0 & \hat{S}_{C,22}^{(3,2)} & (\hat{S}_{C,52}^{(3,2)})^T & 0 & 0 & 0 & (\hat{S}_{C,72}^{(3,2)})^T \\
\hat{S}_{C,51}^{(3,1)} & \hat{S}_{C,52}^{(3,2)} & \hat{S}_{C,55}^{(2,1)} & 0 & 0 & 0 & (\hat{S}_{C,75}^{(2,1)})^T \\
0 & 0 & 0 & \hat{S}_{C,33}^{(3,3)} & 0 & (\hat{S}_{C,63}^{(3,3)})^T & (\hat{S}_{C,73}^{(3,3)})^T \\
0 & 0 & 0 & 0 & \hat{S}_{C,44}^{(3,4)} & (\hat{S}_{C,64}^{(3,4)})^T & (\hat{S}_{C,74}^{(3,4)})^T \\
0 & 0 & 0 & \hat{S}_{C,63}^{(3,3)} & \hat{S}_{C,64}^{(3,4)} & \hat{S}_{C,66}^{(2,2)} & (\hat{S}_{C,76}^{(2,2)})^T \\
\hline\hline
\hat{S}_{C,71}^{(3,1)} & \hat{S}_{C,72}^{(3,2)} & \hat{S}_{C,75}^{(2,1)} & \hat{S}_{C,73}^{(3,3)} & \hat{S}_{C,74}^{(3,4)} & \hat{S}_{C,76}^{(2,2)} & \hat{S}_{C,77}^{(1,1)}
\end{array}\right) \\
&= \left(\begin{array}{cc||c}
A_{55}^{(2,1)} & 0 & (A_{75}^{(2,1)})^T \\
0 & A_{66}^{(2,2)} & (A_{76}^{(2,2)})^T \\
\hline\hline
A_{75}^{(2,1)} & A_{76}^{(2,2)} & A_{77}^{(1,1)}
\end{array}\right) \\
&\equiv \left(\begin{array}{c||c}
A_X & A_Y^T \\
\hline\hline
A_Y & A_Z
\end{array}\right),
\end{aligned}
\tag{2.95}
$$

donde los nodos rechazados como resultado de procesar los subgrafos independientes $G_{(3,1)}$ y $G_{(3,2)}$ se incorporan al separador $(2,1)$, formando los bloques

$$A_{55}^{(2,1)},\ A_{75}^{(2,1)},\ (A_{75}^{(2,1)})^T,$$

y los de $G_{(3,3)}$ y $G_{(3,4)}$ se incorporan al separador $(2,2)$, formando los bloques

$$A_{66}^{(2,2)},\ A_{76}^{(2,2)},\ (A_{77}^{(2,2)})^T.$$

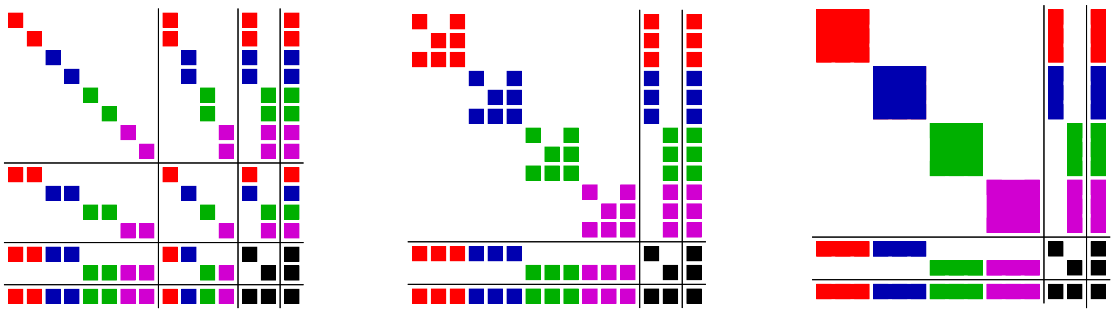
La Figura 2.13 presenta el cálculo de Φ en un caso más general, con $l = 4$ niveles de recursividad. La parte (a) de esta figura muestra la estructura del complemento de Schur aproximado resultante de la primera fase, $\hat{S}_C$, y la parte (c), la estructura de la matriz de entrada, A, para la siguiente fase. La matriz de permutación Φ se obtiene a partir de la estructura de $\hat{S}_C$ de la forma especificada a continuación:

- En primer lugar, se agrupan los nodos de cada par de subgrafos independientes con los nodos del separador del nivel anterior correspondiente. Esta agrupación se representa a través de los bloques del mismo color en la Figura 2.13 (a).

- Seguidamente, se ordena localmente cada grupo, numerando, en primer lugar, los nodos de los subgrafos independientes y, a continuación, los nodos del separador del nivel anterior correspondiente. Como resultado de esta numeración, se genera la estructura de los bloques del mismo color en la Figura 2.13 (b).

- Para terminar, se ordenan globalmente estos grupos y los separadores correspondientes al resto de niveles de forma acorde a la numeración por niveles; ver Figura 2.11. Como resultado de esta numeración, se genera la estructura global de la reordenación, $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$, mostrada en la Figura 2.13 (b).

Al aplicar Φ sobre $\hat{S}_C$ se obtiene una ordenación para la matriz de entrada, A, que permite a la siguiente fase del algoritmo avanzar con el tratamiento de los elementos pivote correspondientes a $\hat{S}_{C_Z}$ y, al mismo tiempo, seguir aprovechando concurrencia durante la construcción del precondicionador en niveles subsecuentes.



(a) (b) (c)

Figura 2.13: (a) Estructura del complemento de Schur aproximado, $\hat{S}_C$, resultante de la primera fase con $l = 4$ niveles de recursividad. (b) Estructura de la reordenación $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$. (c) Matriz de entrada, $A \equiv \Phi^T \hat{S}_C \Phi$, para la segunda fase.

Procesamiento de los separadores del segundo nivel

La segunda fase del algoritmo procede recursivamente con el siguiente nivel, aplicando sobre (2.95) una estrategia *análoga* a la de la primera fase. Por lo tanto, la factorización LDL^T con pivotamiento basado en la inversa factoriza A_X y los bloques no diagonales correspondientes, A_Y e A_Y^T, de forma acorde al núcleo de factorización del Apartado 2.2.2, y no considera los elementos pivote correspondientes a A_Z, obteniendo la siguiente factorización incompleta por bloques

$$\left(\begin{array}{c||c} P_X & 0 \\ \hline\hline 0 & I \end{array}\right)^T \left(\begin{array}{c||c} A_X & A_Y^T \\ \hline\hline A_Y & A_Z \end{array}\right) \left(\begin{array}{c||c} P_X & 0 \\ \hline\hline 0 & I \end{array}\right) = \left(\begin{array}{cc||c} B_X & F_X^T & F_Y^T \\ F_X & C_X & C_Y^T \\ \hline\hline F_Y & C_Y & C_Z \end{array}\right) = \tag{2.96}$$

$$\left(\begin{array}{cc||c} \tilde{L}_{B_X} & 0 & 0 \\ \tilde{L}_{F_X} & I & 0 \\ \hline\hline \tilde{L}_{F_Y} & 0 & I \end{array}\right) \left(\begin{array}{cc||c} \tilde{D}_{B_X} & 0 & 0 \\ 0 & \hat{S}_{C_X} & \hat{S}_{C_Y}^T \\ \hline\hline 0 & \hat{S}_{C_Y} & \hat{S}_{C_Z} \end{array}\right) \left(\begin{array}{cc||c} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T & \tilde{L}_{F_Y}^T \\ 0 & I & 0 \\ \hline\hline 0 & 0 & I \end{array}\right) + \left(\begin{array}{c||c} R_X & R_Y^T \\ \hline\hline R_Y & 0 \end{array}\right),$$

donde los bloques de la matriz resultante de aplicar la estrategia de selección de malla gruesa se definen como

$$B_X = \begin{pmatrix} B_{55}^{(2,1)} & 0 \\ 0 & B_{66}^{(2,2)} \end{pmatrix}, \; F_X = \begin{pmatrix} F_{55}^{(2,1)} & 0 \\ 0 & F_{66}^{(2,2)} \end{pmatrix}, \; F_Y = \begin{pmatrix} F_{75}^{(2,1)} & F_{76}^{(2,2)} \end{pmatrix},$$

$$\left(\begin{array}{c||c} C_X & C_Y^T \\ \hline\hline C_Y & C_Z \end{array}\right) = \left(\begin{array}{cc||c} C_{55}^{(2,1)} & 0 & (C_{75}^{(2,1)})^T \\ 0 & C_{66}^{(2,2)} & (C_{76}^{(2,2)})^T \\ \hline\hline C_{75}^{(2,1)} & C_{76}^{(2,2)} & C_{77}^{(1,1)} \end{array}\right),$$

y los bloques de los factores incompletos y del complemento de Schur aproximado conforme a

$$\tilde{L}_{B_X} = \begin{pmatrix} \tilde{L}_{B,55}^{(2,1)} & 0 \\ 0 & \tilde{L}_{B,66}^{(2,2)} \end{pmatrix}, \; \tilde{L}_{F_X} = \begin{pmatrix} \tilde{L}_{F,55}^{(2,1)} & 0 \\ 0 & \tilde{L}_{F,66}^{(2,2)} \end{pmatrix}, \; \tilde{L}_{F_Y} = \begin{pmatrix} \tilde{L}_{F,75}^{(2,1)} & \tilde{L}_{F,76}^{(2,2)} \end{pmatrix},$$

$$\tilde{D}_{B_X} = \begin{pmatrix} \tilde{D}_{B,55}^{(2,1)} & 0 \\ 0 & \tilde{D}_{B,66}^{(2,2)} \end{pmatrix},$$

$$\left(\begin{array}{c||c} \hat{S}_{C_X} & \hat{S}_{C_Y}^T \\ \hline\hline \hat{S}_{C_Y} & \hat{S}_{C_Z} \end{array}\right) = \left(\begin{array}{cc||c} \hat{S}_{C,55}^{(2,1)} & 0 & (\hat{S}_{C,75}^{(2,1)})^T \\ 0 & \hat{S}_{C,66}^{(2,2)} & (\hat{S}_{C,76}^{(2,2)})^T \\ \hline\hline \hat{S}_{C,75}^{(2,1)} & \hat{S}_{C,76}^{(2,2)} & \hat{S}_{C,77}^{(1,1)} \end{array}\right). \tag{2.97}$$

Los bloques $B_{55}^{(2,1)}$ y $B_{66}^{(2,2)}$, y $C_{55}^{(2,1)}$ y $C_{66}^{(2,2)}$ se corresponden, respectivamente, con los elementos pivote aceptados y rechazados de $A_{55}^{(2,1)}$ y $A_{66}^{(2,1)}$.

La estructura de la matriz (2.95) permite extraer concurrencia en el cálculo de la factorización (2.96). En primer lugar, los bloques diagonales de A_X, $A_{jj}^{(2,i)}$, con $i = 1, 2$, $j = i + 4$, se pueden factorizar independientemente conforme a (2.88):

$$\begin{aligned}(P_{jj}^{(2,i)})^T A_{jj}^{(2,i)} P_{jj}^{(2,i)} &= \begin{pmatrix} B_{jj}^{(2,i)} & (F_{jj}^{(2,i)})^T \\ F_{jj}^{(2,i)} & C_{jj}^{(2,i)} \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{B,jj}^{(2,i)} & 0 \\ \tilde{L}_{F,jj}^{(2,i)} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B,jj}^{(2,i)} & 0 \\ 0 & \hat{S}_{C,jj}^{(2,i)} \end{pmatrix} \begin{pmatrix} (\tilde{L}_{B,jj}^{(2,i)})^T & (\tilde{L}_{F,jj}^{(2,i)})^T \\ 0 & I \end{pmatrix} + R_{jj}^{(2,i)}.\end{aligned} \tag{2.98}$$

En segundo lugar, los bloques de A_Y, $A_{7j}^{(2,i)}$, con $i = 1, 2$, $j = i + 4$, se pueden aproximar de modo independiente como (2.90)

$$\begin{aligned}A_{7j}^{(2,i)} P_{jj}^{(2,i)} &= \begin{pmatrix} F_{7j}^{(2,i)} & C_{7j}^{(2,i)} \end{pmatrix} \\ &= \begin{pmatrix} \tilde{L}_{F,7j}^{(2,i)} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_{B,jj}^{(2,i)} & 0 \\ 0 & \hat{S}_{C,7j}^{(2,i)} \end{pmatrix} \begin{pmatrix} (\tilde{L}_{B,jj}^{(2,i)})^T & (\tilde{L}_{F,jj}^{(2,i)})^T \\ 0 & I \end{pmatrix} + R_{7j}^{(2,i)}.\end{aligned} \tag{2.99}$$

Por último, en la construcción del bloque

$$\begin{aligned}\hat{S}_{C_Z} &= C_Z - \begin{pmatrix} \tilde{L}_{F,75}^{(2,1)} & \tilde{L}_{F,76}^{(2,2)} \end{pmatrix} \tilde{D}_{B_X} \begin{pmatrix} (\tilde{L}_{F,75}^{(2,1)})^T \\ (\tilde{L}_{F,76}^{(2,2)})^T \end{pmatrix} \\ &= C_Z - \left[\left(\tilde{L}_{F,75}^{(2,1)} \tilde{D}_{B,55}^{(2,1)} (\tilde{L}_{F,75}^{(2,1)})^T \right) + \left(\tilde{L}_{F,76}^{(2,2)} \tilde{D}_{B,66}^{(2,2)} (\tilde{L}_{F,76}^{(2,2)})^T \right) \right],\end{aligned} \tag{2.100}$$

ambos sumandos se pueden calcular de forma independiente, aunque su acumulación requiere una sincronización (reducción); ver Apartado 2.3.3. Además, el grado del paralelismo que se puede explotar en esta fase se reduce a la mitad con respecto al de la primera.

La segunda fase del algoritmo procede recursivamente con el siguiente nivel repitiendo la factorización (2.96) con $A \equiv \hat{S}_C$, hasta que finalmente el tamaño del bloque $\hat{S}_{C_X}$ en (2.97) es "suficientemente pequeño". Las computaciones detalladas hasta el momento para la segunda fase se expresan en el Algoritmo 2.17. Como se puede observar allí, la matriz de entrada se obtiene a partir de la reordenación $\hat{S}_C \rightarrow \Phi \hat{S}_C \Phi$ del complemento de Schur aproximado resultante de la primera fase; ver Figura 2.13.

Algoritmo 2.17: Segunda fase de la transformación del Algoritmo 2.15.

Entrada: $A \equiv \Phi^T \hat{S}_C \Phi \equiv \left(\begin{array}{c||c} A_X & A_Y^T \\ \hline\hline A_Y & A_Z \end{array} \right)$

1. Calcular la factorización incompleta por bloques (2.96) del bloque A_X, utilizando el núcleo de factorización LDL^T descrito en el Apartado 2.2.2.

2. Proceder con el siguiente nivel repitiendo el paso 1 con

$$A \equiv \hat{S}_C \equiv \left(\begin{array}{c||c} A_X & A_Y^T \\ \hline\hline A_Y & A_Z \end{array} \right),$$

hasta que finalmente $\hat{S}_{C_X}$ en (2.96) quede vacío o sea lo "suficientemente pequeño".

Tras efectuar las computaciones de la segunda fase, el algoritmo entra finalmente en su tercera y última fase. La matriz de entrada de la última fase, se obtiene a partir de una reordenación

de (2.97), $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$, que, en el caso de la última fase, equivale a $\hat{S}_C$

$$A \equiv \Phi^T \hat{S}_C \Phi \equiv \hat{S}_C = \begin{pmatrix} \hat{S}_{C,55}^{(2,1)} & 0 & (\hat{S}_{C,75}^{(2,1)})^T \\ 0 & \hat{S}_{C,66}^{(2,2)} & (\hat{S}_{C,76}^{(2,2)})^T \\ \hat{S}_{C,75}^{(2,1)} & \hat{S}_{C,76}^{(2,2)} & \hat{S}_{C,77}^{(1,1)} \end{pmatrix}. \tag{2.101}$$

Procesamiento del separador del primer nivel

Las computaciones de la última fase se expresan en el Algoritmo 2.18. Como se puede observar allí, la matriz de entrada se obtiene a partir de la reordenación $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$ del complemento de Schur aproximado resultante de la segunda fase; ver (2.101). A diferencia de los Algoritmos 2.16 y 2.17, la factorización incompleta por bloques del paso 1 trata todos los elementos pivote de A, y el paso 2 repite el paso 1 con $A \equiv \hat{S}_C$, hasta que finalmente $\hat{S}_C$ quede vacío o se vuelva "suficientemente denso". En este segundo caso, el paso 3 invoca a un núcleo de factorización directa para matrices densas, y se puede explotar potencialmente paralelismo para el cálculo de la factorización $\hat{S}_C = L_C D_C L_C^T$ utilizando, por ejemplo, la biblioteca LAPACK y una versión multihebra de la biblioteca BLAS.

Algoritmo 2.18: Tercera fase de la transformación del Algoritmo 2.15.

Entrada: $A \equiv \Phi^T \hat{S}_C \Phi = \hat{S}_C$

1. Calcular la factorización incompleta por bloques

$$P^T A P = \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} + R, \tag{2.102}$$

   utilizando el núcleo de factorización LDL^T descrito en el Apartado 2.2.2.

2. Proceder con el siguiente nivel repitiendo el paso 1 con $A \equiv \hat{S}_C$, hasta que finalmente $\hat{S}_C$ quede vacío o se vuelva lo "suficientemente denso" como para ser gestionado eficientemente mediante un núcleo de factorización directa para matrices densas.

3. Si $\hat{S}_C$ no es un bloque vacío, calcular su factorización $\hat{S}_C = L_C D_C L_C^T$, utilizando un método directo para matrices densas.

Definición de la transformación del algoritmo básico

Integrando los Algoritmos 2.16, 2.17 y 2.18 en un sólo algoritmo, y generalizando el proceso descrito en este apartado para un número arbitrario de niveles l, obtendríamos finalmente la transformación del Algoritmo 2.15; ver Algoritmo 2.19. El algoritmo incluye, en los pasos 1 y 2, la aplicación del escalado y de la permutación de reducción de llenado, respectivamente. Estas computaciones se restringen a los bloques A_X y $\tilde{A}_X$, respectivamente, por las mismas razones que lo hace la factorización con pivotamiento basado en la inversa en el paso 3. En la fase l-ésima, el tamaño del bloque A_Z en el particionado de A es nulo y, por lo tanto, los pasos 1, 2 y 3 se aplican sobre todas las filas y columnas de las respectivas matrices de entrada. El paso 4 se obtiene a partir del paso 2 de los Algoritmos 2.16, 2.17 y 2.18. El paso 5 prepara la matriz de entrada para la siguiente fase, obteniendo la reordenación $\hat{S}_C \rightarrow \Phi^T \hat{S}_C \Phi$ del complemento de Schur aproximado resultante de la fase actual y, por último, el paso 6 se obtiene a partir del paso 3 del Algoritmo 2.18. *El Algoritmo 2.19 permite explotar paralelismo de grado 2^{l-1} en la primera fase, paralelismo de grado 2^{l-2} en la segunda, y así sucesivamente, hasta la última fase, en la que no hay paralelismo que explotar de este tipo.*

Algoritmo 2.19: Transformación del Algoritmo 2.15.

Entrada: $A \equiv \Omega^T A \Omega \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right)$

para *fase*$= 1, \ldots, l$ **hacer**

1. Calcular y aplicar al bloque A_X la matriz de escalado diagonal,

$$\left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right) \left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right) = \left(\begin{array}{c|c} \tilde{A}_X & \tilde{A}_Y^T \\ \hline \tilde{A}_Y & A_Z \end{array}\right).$$

2. Calcular y aplicar al bloque $\tilde{A}_X$ la permutación para reducción de llenado,

$$\left(\begin{array}{c|c} \hat{P}_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c|c} \tilde{A}_X & \tilde{A}_Y^T \\ \hline \tilde{A}_Y & A_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} \hat{P}_X & 0 \\ \hline 0 & I \end{array}\right)^T \left(\begin{array}{c|c} \tilde{A}_X & \tilde{A}_Y^T \\ \hline \tilde{A}_Y & A_Z \end{array}\right) \left(\begin{array}{c|c} \hat{P}_X & 0 \\ \hline 0 & I \end{array}\right) = \left(\begin{array}{c|c} \hat{A}_X & \hat{A}_Y^T \\ \hline \hat{A}_Y & A_Z \end{array}\right).$$

3. Calcular la factorización incompleta por bloques del bloque $\hat{A}_X$,

$$\left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right)^T \left(\begin{array}{c|c} \hat{A}_X & \hat{A}_Y^T \\ \hline \hat{A}_Y & A_Z \end{array}\right) \left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right) = \left(\begin{array}{cc|c} B_X & F_X^T & F_Y^T \\ F_X & C_X & C_Y^T \\ \hline F_Y & C_Y & C_Z \end{array}\right) = \tag{2.103}$$

$$\left(\begin{array}{cc|c} \tilde{L}_{B_X} & 0 & 0 \\ \tilde{L}_{F_X} & I & 0 \\ \hline \tilde{L}_{F_Y} & 0 & I \end{array}\right) \left(\begin{array}{cc|c} \tilde{D}_{B_X} & 0 & 0 \\ 0 & \hat{S}_{C_X} & \hat{S}_{C_Y}^T \\ \hline 0 & \hat{S}_{C_Y} & \hat{S}_{C_Z} \end{array}\right) \left(\begin{array}{cc|c} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T & \tilde{L}_{F_Y}^T \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right) + \left(\begin{array}{c|c} R_X & R_Y^T \\ \hline R_Y & 0 \end{array}\right),$$

utilizando el núcleo de factorización LDL^T descrito en el Apartado 2.2.2.

4. Proceder con el siguiente nivel repitiendo los pasos 1, 2 y 3 con

$$A \equiv \hat{S}_C \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right),$$

hasta que finalmente $\hat{S}_{C_X}$ sea un bloque vacío o sea lo "suficientemente pequeño", si *fase* $< l$, o hasta que $\hat{S}_C$ desaparezca o lo "suficientemente denso", si *fase* $= l$.

si *fase* $< l$ **entonces**

5. Proceder a la siguiente fase con

$$A \equiv \Phi^T \hat{S}_C \Phi \equiv \left(\begin{array}{c|c} A_X & A_Y^T \\ \hline A_Y & A_Z \end{array}\right).$$

si no

6. Si $\hat{S}_C$ no es un bloque vacío, calcular la factorización $\hat{S}_C = L_C D_C L_C^T$ utilizando un método directo para matrices densas.

fin si

fin para

2.3.3. Precondicionadores paralelos basados en la factorización LDL^T multinivel

El algoritmo paralelo que construye el precondicionador multinivel explota la concurrencia que se pone de manifiesto en las computaciones que efectúa el Algoritmo 2.19. Dicha concurrencia se captura a través del árbol de dependencias entre tareas que se muestra en la Figura 2.14. Este árbol, que se obtiene como resultado del proceso de disección anidada, descompone las computaciones del Algoritmo 2.19 en *tareas* (nodos del árbol), y guía su ejecución concurrente identificando las *dependencias* existentes entre las mismas (arcos entre nodos del árbol). Nótese que no todas las tareas del árbol son independientes entre sí, dado que algunas tareas usan los resultados de las computaciones que producen otras y, por lo tanto, tienen que esperar a que éstas últimas los produzcan. Si en el árbol de dependencias entre tareas hay un arco del nodo i al j, entonces las computaciones que lleva a cabo el nodo j no pueden iniciarse hasta que hayan finalizado las de i, es decir, j *depende de* i. Además, diremos que j es el padre de i, y que i es el hijo de j. Por ejemplo, en la Figura 2.14, $(1,1)$ es el padre de $(2,1)$ y $(2,2)$, y $(2,1)$ y $(2,2)$ son los hijos de $(1,1)$.

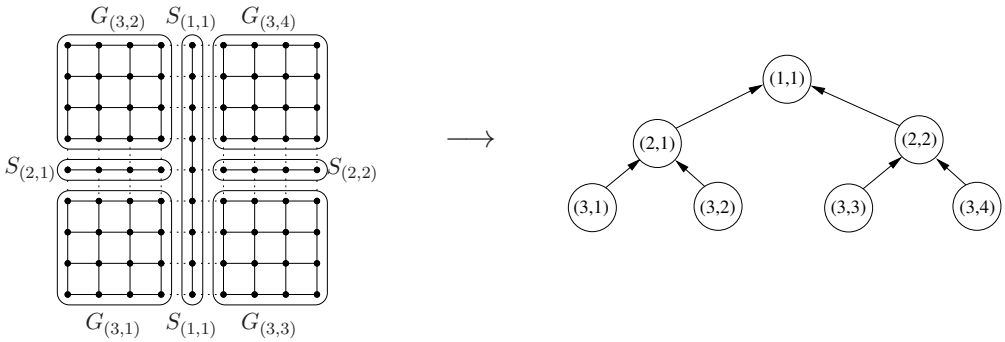


Figura 2.14: Disección anidada (izquierda) y árbol de dependencias entre tareas (derecha).

Procesamiento paralelo de los subgrafos independientes

El cálculo paralelo del precondicionador se inicia con aquellas tareas que no tienen dependencias, es decir, las tareas hoja del árbol. Estas tareas efectúan, de modo independiente, las computaciones de la *primera fase* del Algoritmo 2.19, explotando para ello el paralelismo que se pone de manifiesto en el cálculo de la factorización parcial (2.93). Cada tarea hoja $(3,i)$, con $i = 1,\ldots,4$, factoriza $A_{ii}^{(3,i)}$, y extiende esta factorización a los bloques de A_Y e A_Y^T, conforme a (2.88) y (2.90), respectivamente. Para facilitar la exposición, nos referiremos a estos bloques como bloques que factoriza la tarea hoja $(3,i)$. Así, los bloques que factoriza la tarea $(3,2)$ son $A_{22}^{(3,2)}$, $A_{52}^{(3,2)}$ y $A_{72}^{(3,2)}$. Para completar (2.93), falta especificar cómo se gestiona la construcción de $\hat{S}_{C_Z}$; ver (2.94). Con este propósito, el algoritmo paralelo tiene en cuenta las siguientes consideraciones:

- Cada tarea hoja *únicamente* contribuye al cálculo de los bloques de $\hat{S}_{C_Z}$ correspondientes a los ancestros situados en el camino del árbol que parte de esa propia tarea hoja y termina en la raíz. Siguiendo el ejemplo de la Figura 2.14, la tarea hoja $(3,2)$ contribuye en la construcción de los bloques correspondientes a los separadores $(2,1)$ y $(1,1)$. Teniendo en cuenta que estos bloques se obtienen como

$$\hat{S}_{C,55}^{(2,1)} = C_{55}^{(2,1)} - \tilde{L}_{F,51}^{(3,1)}\tilde{D}_{B,11}^{(3,1)}(\tilde{L}_{F,51}^{(3,1)})^T - \tilde{\mathbf{L}}_{\mathbf{F,52}}^{\mathbf{(3,2)}}\tilde{\mathbf{D}}_{\mathbf{B,22}}^{\mathbf{(3,2)}}(\tilde{\mathbf{L}}_{\mathbf{F,52}}^{\mathbf{(3,2)}})^{\mathbf{T}}, \quad (2.104)$$

$$\hat{S}_{C,75}^{(2,1)} = C_{75}^{(2,1)} - \tilde{L}_{F,71}^{(3,1)}\tilde{D}_{B,11}^{(3,1)}(\tilde{L}_{F,51}^{(3,1)})^T - \tilde{\mathbf{L}}_{\mathbf{F,72}}^{\mathbf{(3,2)}}\tilde{\mathbf{D}}_{\mathbf{B,22}}^{\mathbf{(3,2)}}(\tilde{\mathbf{L}}_{\mathbf{F,52}}^{\mathbf{(3,2)}})^{\mathbf{T}} \text{ y} \quad (2.105)$$

$$(\hat{S}_{C,75}^{(2,1)})^T = (C_{75}^{(2,1)})^T - \tilde{L}_{F,51}^{(3,1)}\tilde{D}_{B,11}^{(3,1)}(\tilde{L}_{F,71}^{(3,1)})^T - \tilde{\mathbf{L}}_{\mathbf{F,52}}^{\mathbf{(3,2)}}\tilde{\mathbf{D}}_{\mathbf{B,22}}^{\mathbf{(3,2)}}(\tilde{\mathbf{L}}_{\mathbf{F,72}}^{\mathbf{(3,2)}})^{\mathbf{T}}, \quad (2.106)$$

y

$$\begin{aligned}\hat{S}^{(1,1)}_{C,77} = C^{(1,1)}_{77} &- \tilde{L}^{(3,1)}_{F,71}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,71})^T - \mathbf{\tilde{L}^{(3,2)}_{F,72}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,72})^T} \\ &- \tilde{L}^{(3,3)}_{F,73}\tilde{D}^{(3,3)}_{B,33}(\tilde{L}^{(3,3)}_{F,73})^T - \tilde{L}^{(3,4)}_{F,74}\tilde{D}^{(3,4)}_{B,44}(\tilde{L}^{(3,4)}_{F,74})^T, \end{aligned} \quad (2.107)$$

se han marcado en negrita las contribuciones de la tarea hoja $(3,2)$. De aquí en adelante, nos referiremos a los bloques de $\hat{S}_{C_Z}$ como *bloques de contribución*.

- No es necesario calcular $\hat{S}_{C_Z}$ *explícitamente* mientras el algoritmo no proceda con su factorización. En su lugar, las tareas hoja pueden calcular y almacenar local e independientemente sus contribuciones a los bloques de $\hat{S}_{C_Z}$, y sólo acumular estas contribuciones para recuperar las entradas de $\hat{S}_{C_Z}$ en el momento en que el algoritmo proceda con su factorización. Por ejemplo, consideremos el cálculo del bloque (2.104). Si descomponemos el bloque $C^{(2,1)}_{55}$ como

$$C^{(2,1)}_{55} = C^{(3,1)}_{55} + C^{(3,2)}_{55}, \quad (2.108)$$

donde $C^{(3,1)}_{55}$ y $C^{(3,2)}_{55}$ son, respectivamente, bloques locales de las tareas $(3,1)$ y $(3,2)$ correspondientes a su ancestro $(2,1)$, entonces la tarea $(3,1)$ puede calcular sus contribuciones locales a $\hat{S}^{(2,1)}_{C,55}$ como

$$\hat{S}^{(3,1)}_{C,55} = C^{(3,1)}_{55} - \tilde{L}^{(3,1)}_{F,51}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,51})^T,$$

y la tarea $(3,2)$ puede hacer lo propio como

$$\hat{S}^{(3,2)}_{C,55} = C^{(3,2)}_{55} - \tilde{L}^{(3,2)}_{F,52}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,52})^T.$$

Cuando el algoritmo paralelo requiera la factorización de $\hat{S}^{(2,1)}_{C,55}$, las entradas de este bloque se pueden recuperar acumulando las contribuciones locales de las tareas $(3,1)$ y $(3,2)$; es decir,

$$\begin{aligned}\hat{S}^{(2,1)}_{C,55} &= \hat{S}^{(3,1)}_{C,55} + \hat{S}^{(3,2)}_{C,55} \\ &= C^{(3,1)}_{55} - \tilde{L}^{(3,1)}_{F,51}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,51})^T + C^{(3,2)}_{55} - \tilde{L}^{(3,2)}_{F,52}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,52})^T \\ &= C^{(2,1)}_{55} - \tilde{L}^{(3,1)}_{F,51}\tilde{D}^{(3,1)}_{B,11}(\tilde{L}^{(3,1)}_{F,51})^T - \tilde{L}^{(3,2)}_{F,52}\tilde{D}^{(3,2)}_{B,22}(\tilde{L}^{(3,2)}_{F,52})^T. \end{aligned} \quad (2.109)$$

De aquí en adelante, nos referiremos a los bloques $\hat{S}^{(3,1)}_{C,55}$ y $\hat{S}^{(3,2)}_{C,55}$, y los obtenidos para el cálculo del resto de bloques de contribución, como *bloques locales de contribución*.

- La consideración anterior es válida para cualquier descomposición del bloque $C^{(2,1)}_{55}$ que cumpla (2.108). Entre todas las opciones posibles, hemos escogido arbitrariamente aquella para la cual los términos de la suma (2.108) contribuyen aritméticamente en la misma proporción a $C^{(2,1)}_{55}$. En otras palabras,

$$\frac{1}{2}C^{(2,1)}_{55} = C^{(3,1)}_{55} = C^{(3,2)}_{55}.$$

En el caso del bloque $C^{(1,1)}_{77}$, la descomposición necesaria para explotar la consideración anterior es

$$C^{(1,1)}_{77} = C^{(3,1)}_{77} + C^{(3,2)}_{77} + C^{(3,3)}_{77} + C^{(3,4)}_{77},$$

y, por tanto,

$$\frac{1}{4}C^{(1,1)}_{77} = C^{(3,1)}_{77} = C^{(3,2)}_{77} = C^{(3,3)}_{77} = C^{(3,4)}_{77}.$$

Teniendo en cuenta las consideraciones anteriores, si queremos calcular $\hat{S}_{C_Z}$, sólo hace falta descomponer C_Z en los siguientes sumandos,

$$\begin{aligned}
C_Z &= \left(\begin{array}{cc||c} C_{55}^{(2,1)} & 0 & (C_{75}^{(2,1)})^T \\ 0 & C_{66}^{(2,2)} & (C_{76}^{(2,2)})^T \\ \hline\hline C_{75}^{(2,1)} & C_{76}^{(2,2)} & C_{77}^{(1,1)} \end{array}\right) \\
&= \left(\begin{array}{cc||c} C_{55}^{(3,1)} & 0 & (C_{75}^{(3,1)})^T \\ 0 & 0 & 0 \\ \hline\hline C_{75}^{(3,1)} & 0 & C_{77}^{(3,1)} \end{array}\right) + \left(\begin{array}{cc||c} C_{55}^{(3,2)} & 0 & (C_{75}^{(3,2)})^T \\ 0 & 0 & 0 \\ \hline\hline C_{75}^{(3,2)} & 0 & C_{77}^{(3,2)} \end{array}\right) + \\
&\quad \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & C_{66}^{(3,3)} & (C_{76}^{(3,3)})^T \\ \hline\hline 0 & C_{76}^{(3,3)} & C_{77}^{(3,3)} \end{array}\right) + \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & C_{66}^{(3,4)} & (C_{76}^{(3,4)})^T \\ \hline\hline 0 & C_{76}^{(3,4)} & C_{77}^{(3,4)} \end{array}\right) \\
&= C_Z^{(3,1)} + C_Z^{(3,2)} + C_Z^{(3,3)} + C_Z^{(3,4)},
\end{aligned} \tag{2.110}$$

donde $C_Z^{(3,i)}$, con $i = 1, \ldots, 4$, contiene los bloques locales de la tarea hoja $(3, i)$ correspondientes a sus ancestros. Substituyendo esta ecuación en (2.94), se obtiene finalmente que

$$\begin{aligned}
\hat{S}_{C_Z} &= \left(\begin{array}{cc||c} \hat{S}_{C,55}^{(2,1)} & 0 & (\hat{S}_{C,75}^{(2,1)})^T \\ 0 & \hat{S}_{C,66}^{(2,2)} & (\hat{S}_{C,76}^{(2,2)})^T \\ \hline\hline \hat{S}_{C,75}^{(2,1)} & \hat{S}_{C,76}^{(2,2)} & \hat{S}_{C,77}^{(1,1)} \end{array}\right) \\
&= \left[\left(\begin{array}{cc||c} C_{55}^{(3,1)} & 0 & (C_{75}^{(3,1)})^T \\ 0 & 0 & 0 \\ \hline\hline C_{75}^{(3,1)} & 0 & C_{77}^{(3,1)} \end{array}\right) - \left(\begin{array}{cc||c} \tilde{L}_{F,51}^{(3,1)} \tilde{D}_{B,11}^{(3,1)} (\tilde{L}_{F,51}^{(3,1)})^T & 0 & \tilde{L}_{F,51}^{(3,1)} \tilde{D}_{B,11}^{(3,1)} (\tilde{L}_{F71}^{(3,1)})^T \\ 0 & 0 & 0 \\ \hline\hline \tilde{L}_{F,71}^{(3,1)} \tilde{D}_{B,11}^{(3,1)} (\tilde{L}_{F,51}^{(3,1)})^T & 0 & \tilde{L}_{F,71}^{(3,1)} \tilde{D}_{B,11}^{(3,1)} (\tilde{L}_{F,71}^{(3,1)})^T \end{array}\right)\right] \\
&+ \left[\left(\begin{array}{cc||c} C_{55}^{(3,2)} & 0 & (C_{75}^{(3,2)})^T \\ 0 & 0 & 0 \\ \hline\hline C_{75}^{(3,2)} & 0 & C_{77}^{(3,2)} \end{array}\right) - \left(\begin{array}{cc||c} \tilde{L}_{F,52}^{(3,2)} \tilde{D}_{B,22}^{(3,2)} (\tilde{L}_{F,52}^{(3,2)})^T & 0 & \tilde{L}_{F,52}^{(3,2)} \tilde{D}_{B,22}^{(3,2)} (\tilde{L}_{F,72}^{(3,2)})^T \\ 0 & 0 & 0 \\ \hline\hline \tilde{L}_{F,72}^{(3,2)} \tilde{D}_{B,22}^{(3,2)} (\tilde{L}_{F,52}^{(3,2)})^T & 0 & \tilde{L}_{F,72}^{(3,2)} \tilde{D}_{B,22}^{(3,2)} (\tilde{L}_{F,72}^{(3,2)})^T \end{array}\right)\right] \\
&+ \left[\left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & C_{66}^{(3,3)} & (C_{76}^{(3,3)})^T \\ \hline\hline 0 & C_{76}^{(3,3)} & C_{77}^{(3,3)} \end{array}\right) - \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & \tilde{L}_{F,63}^{(3,3)} \tilde{D}_{B,33}^{(3,3)} (\tilde{L}_{F,63}^{(3,3)})^T & \tilde{L}_{F,63}^{(3,3)} \tilde{D}_{B,33}^{(3,3)} (\tilde{L}_{F,73}^{(3,3)})^T \\ \hline\hline 0 & \tilde{L}_{F,73}^{(3,3)} \tilde{D}_{B,33}^{(3,3)} (\tilde{L}_{F,63}^{(3,3)})^T & \tilde{L}_{F,73}^{(3,3)} \tilde{D}_{B,33}^{(3,3)} (\tilde{L}_{F,73}^{(3,3)})^T \end{array}\right)\right] \\
&+ \left[\left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & C_{66}^{(3,4)} & (C_{76}^{(3,4)})^T \\ \hline\hline 0 & C_{76}^{(3,4)} & C_{77}^{(3,4)} \end{array}\right) - \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & \tilde{L}_{F,64}^{(3,4)} \tilde{D}_{B,44}^{(3,4)} (\tilde{L}_{F,64}^{(3,4)})^T & \tilde{L}_{F,64}^{(3,4)} \tilde{D}_{B,44}^{(3,4)} (\tilde{L}_{F,74}^{(3,4)})^T \\ \hline\hline 0 & \tilde{L}_{F,74}^{(3,4)} \tilde{D}_{B,44}^{(3,4)} (\tilde{L}_{F,64}^{(3,4)})^T & \tilde{L}_{F,74}^{(3,4)} \tilde{D}_{B,44}^{(3,4)} (\tilde{L}_{F,74}^{(3,4)})^T \end{array}\right)\right] \\
&= \hat{S}_{C_Z}^{(3,1)} + \hat{S}_{C_Z}^{(3,2)} + \hat{S}_{C_Z}^{(3,3)} + \hat{S}_{C_Z}^{(3,4)},
\end{aligned} \tag{2.111}$$

donde $\hat{S}_{C_Z}^{(3,i)}$, con $i = 1, \ldots, 4$, contiene los bloques locales de contribución calculados por la tarea hoja $(3, i)$ de forma independiente al resto. Por esta razón, si mantenemos los bloques locales de contribución en cada tarea hoja, entonces se puede calcular local e independientemente $\hat{S}_{C_Z}^{(3,i)}$, evitando puntos de sincronización mientras el algoritmo paralelo no proceda con la factorización de $\hat{S}_{C_Z}$, aumentando de forma significativa el grado de concurrencia.

Por todo lo anterior, si queremos explotar el paralelismo de la forma descrita, debemos descomponer *explícitamente* la matriz de entrada para la primera fase del algoritmo en cuatro sumandos, uno por cada tarea hoja, en los que se incluyen los bloques con los que opera cada una; ver parte superior de la Figura 2.15. Así, cada sumando está compuesto por los bloques que factoriza la tarea hoja, junto con los bloques locales de esta tarea correspondientes a sus

ancestros. Estos bloques se inicializan con las entradas correspondientes de A, divididas por 2^{l-k}, donde l es el número de niveles de la disección anidada, y k es el nivel de la recursividad al que pertenece el separador u hoja correspondiente. Por ejemplo, en la Figura 2.15, los bloques a factorizar por la tarea hoja $(3,2)$, es decir, $A_{22}^{(3,2)}$, $A_{52}^{(3,2)}$ y $A_{72}^{(3,2)}$, se inicializan con las entradas correspondientes de A, divididas por $2^{3-3}=1$; los bloques locales asociados al separador $(2,1)$, es decir, $A_{55}^{(3,2)}$, $A_{75}^{(3,2)}$, se inicializan respectivamente, con las entradas de los bloques $A_{55}^{(2,1)}$, $A_{75}^{(2,1)}$, divididas por $2^{3-2}=2$; y el bloque local de la tarea $(3,2)$ asociado al separador $(1,1)$, es decir, $A_{77}^{(3,2)}$, con las entradas del bloque $A_{77}^{(1,1)}$, divididas por $2^{3-1}=4$. Realmente, la matriz de entrada de cada tarea hoja, $A^{(3,i)}$, con $i=1,\ldots,4$, se forma eliminando los bloques nulos de los sumandos obtenidos al descomponer la matriz A, tal y como aparece en la parte inferior de la Figura 2.15. Además, la opción propuesta para inicializar las entradas de los bloques locales de las tareas no incurre en errores de redondeo ya que, en el sistema de representación de números en coma flotante que emplea el computador, la división entre potencias de dos se implementa con un cambio en su exponente, sin que afecte al valor de la mantisa y por tanto a su precisión.

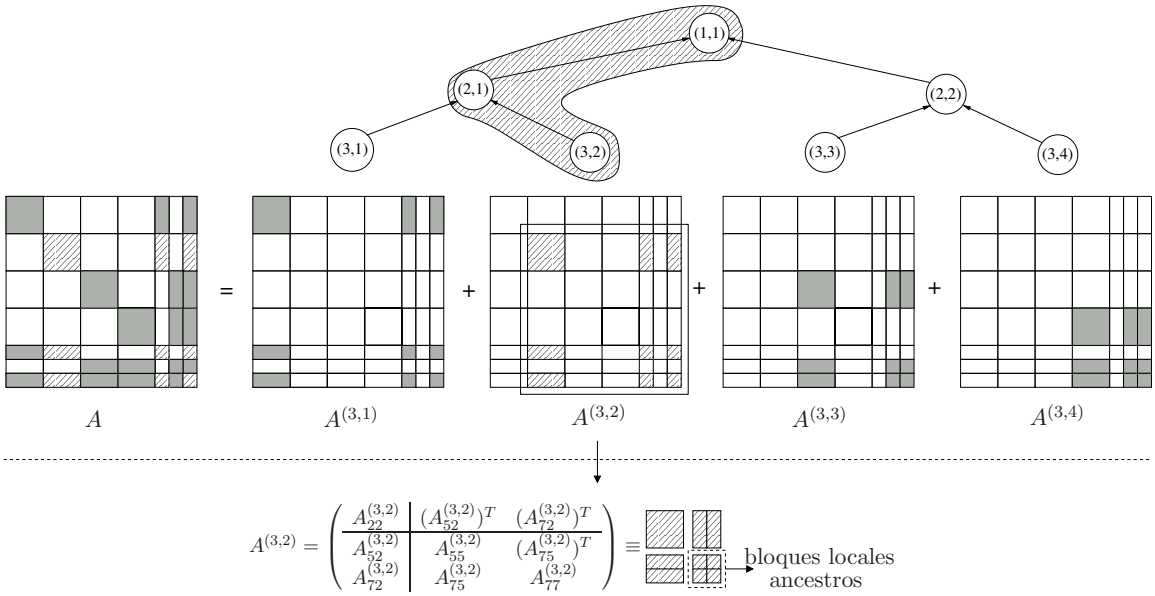


Figura 2.15: Descomposición inicial de la matriz de entrada para la primera fase del Algoritmo 2.19.

Para paralelizar las computaciones de la primera fase del Algoritmo 2.19, cada tarea hoja $(3,i)$ procesa $A^{(3,i)}$, aplicando *localmente* el siguiente algoritmo:

- En los pasos 1 y 2, la matriz de diagonal, $D^{(3,i)}$, $A^{(3,i)} \rightarrow D^{(3,i)}A^{(3,i)}D^{(3,i)} = \tilde{A}^{(3,i)}$, y la permutación para reducción de llenado, $\tilde{P}^{(3,i)}$, $\tilde{A}^{(3,i)} \rightarrow (\tilde{P}^{(3,i)})^T\tilde{A}^{(3,i)}\tilde{P}^{(3,i)} = \hat{A}^{(3,i)}$, se restringen a los bloques que factoriza la tarea. Por lo tanto, si particionamos $D^{(3,i)}$ y $\tilde{P}^{(3,i)}$ conforme con $A^{(3,i)}$, entonces

$$D^{(3,i)} = \left(\begin{array}{c|cc} D_{ii}^{(3,i)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \text{ y } \tilde{P}^{(3,i)} = \left(\begin{array}{c|cc} \tilde{P}_{ii}^{(3,i)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right). \tag{2.112}$$

- En el paso 3, la tarea hoja aplica el núcleo de factorización LDL^T con pivotamiento basado en la inversa a la matriz $\hat{A}^{(3,i)}$. En el proceso de factorización, la estrategia de pivotamiento no considera elementos pivote de los bloques locales correspondientes a los ancestros de la

tarea y, por tanto, la factorización incompleta por bloques que se obtiene como resultado tiene la siguiente estructura (para la tarea $(3,2)$):

$$(P^{(3,2)})^T \hat{A}^{(3,2)} P^{(3,2)} = \left(\begin{array}{cc|cc} B_{22}^{(3,2)} & (F_{22}^{(3,2)})^T & (F_{52}^{(3,2)})^T & (F_{72}^{(3,2)})^T \\ F_{22}^{(3,2)} & C_{22}^{(3,2)} & (C_{52}^{(3,2)})^T & (C_{72}^{(3,2)})^T \\ \hline F_{52}^{(3,2)} & C_{52}^{(3,2)} & C_{55}^{(3,2)} & (C_{75}^{(3,2)})^T \\ F_{72}^{(3,2)} & C_{72}^{(3,2)} & C_{75}^{(3,2)} & C_{77}^{(3,2)} \end{array}\right) \approx \tilde{L}^{(3,2)} \tilde{D}^{(3,2)} (\tilde{L}^{(3,2)})^T, \tag{2.113}$$

con

$$\tilde{L}^{(3,2)} = \left(\begin{array}{cc|cc} \tilde{L}_{B,22}^{(3,2)} & 0 & 0 & 0 \\ \tilde{L}_{F,22}^{(3,2)} & I & 0 & 0 \\ \hline \tilde{L}_{F,52}^{(3,2)} & 0 & I & 0 \\ \tilde{L}_{F,72}^{(3,2)} & 0 & 0 & I \end{array}\right), \quad \tilde{D}^{(3,2)} = \left(\begin{array}{cc|cc} \tilde{D}_{B,22}^{(3,2)} & 0 & 0 & 0 \\ 0 & \hat{S}_{C,22}^{(3,2)} & (\hat{S}_{C,52}^{(3,2)})^T & (\hat{S}_{C,72}^{(3,2)})^T \\ \hline 0 & \hat{S}_{C,52}^{(3,2)} & \hat{S}_{C,55}^{(3,2)} & (\hat{S}_{C,75}^{(3,2)})^T \\ 0 & \hat{S}_{C,72}^{(3,2)} & \hat{S}_{C,75}^{(3,2)} & \hat{S}_{C,77}^{(3,2)} \end{array}\right).$$

Los bloques $B_{22}^{(3,2)}$ y $C_{22}^{(3,2)}$ contienen, respectivamente, los elementos pivote aceptados y rechazados localmente por la tarea $(3,2)$. $C_{55}^{(3,2)}$, $(C_{75}^{(3,2)})^T$, $C_{75}^{(3,2)}$ y $C_{77}^{(3,2)}$ son los bloques locales de $(3,2)$ correspondientes a sus ancestros; ver sumando $C_Z^{(3,2)}$ en (2.110). Por último, $\hat{S}_{C,55}^{(3,2)}$, $(\hat{S}_{C,75}^{(3,2)})^T$, $\hat{S}_{C,75}^{(3,2)}$, y $\hat{S}_{C,77}^{(3,2)}$ son los bloques locales de contribución de $(3,2)$; ver sumando $\hat{S}_{C_Z}^{(3,2)}$ en (2.111). La Figura 2.16 ilustra de forma gráfica el proceso a través del cual la tarea $(3,2)$ calcula la factorización incompleta por bloques (2.113).

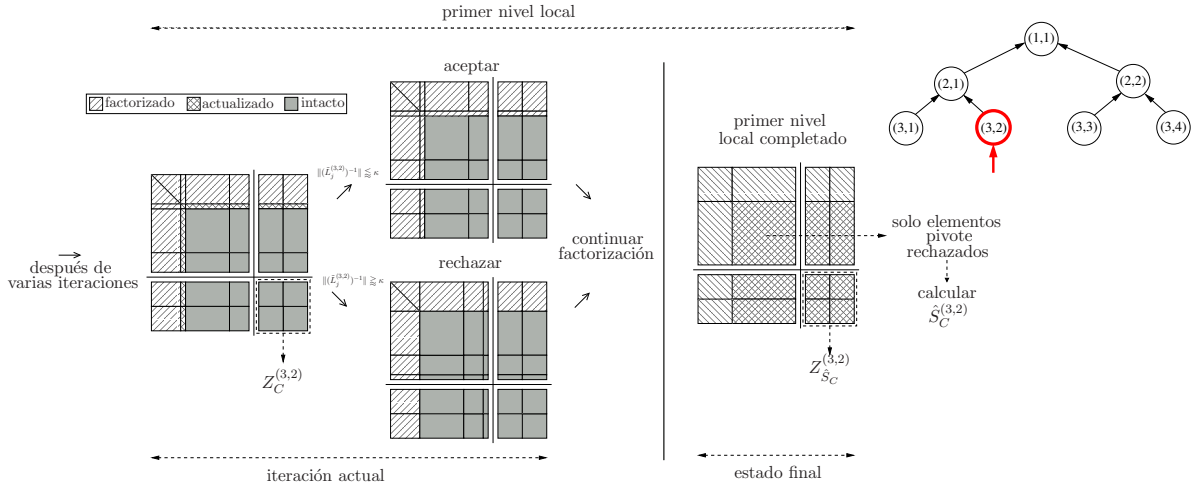


Figura 2.16: Factorización LDL^T con pivotamiento basado en la inversa local correspondiente a la tarea hoja $(3,2)$.

- En el paso 4, la tarea hoja procede recursivamente con el siguiente nivel, repitiendo los pasos 1, 2 y 3 con el complemento de Schur aproximado como matriz de entrada al siguiente nivel; es decir, con

$$A^{(3,2)} \equiv \left(\begin{array}{c|cc} \hat{S}_{C,22}^{(3,2)} & (\hat{S}_{C,52}^{(3,2)})^T & (\hat{S}_{C,72}^{(3,2)})^T \\ \hline \hat{S}_{C,52}^{(3,2)} & \hat{S}_{C,55}^{(3,2)} & (\hat{S}_{C,75}^{(3,2)})^T \\ \hat{S}_{C,72}^{(3,2)} & \hat{S}_{C,75}^{(3,2)} & \hat{S}_{C,77}^{(3,2)} \end{array}\right), \tag{2.114}$$

en el caso de la tarea $(3,2)$, hasta que finalmente el tamaño el bloque correspondiente a los elementos pivote rechazados, es decir, $\hat{S}_{C,22}^{(3,2)}$, es lo "suficientemente pequeño"; ver Fi-

gura 2.17. Finalmente, cada hoja "envía" el complemento de Schur aproximado resultante a su padre; es decir, $(3,2)$ "envía" (2.114) a $(2,1)$.

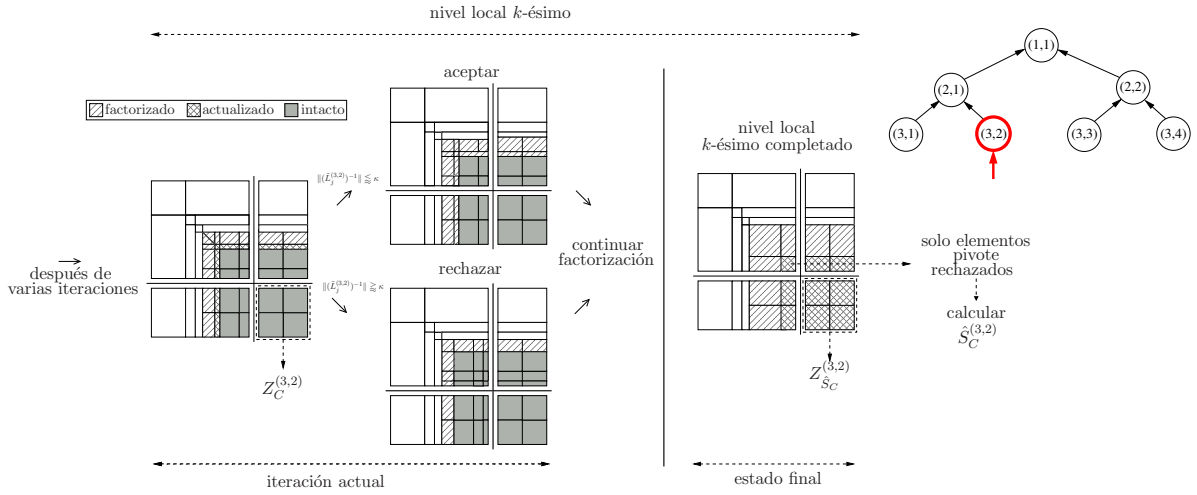


Figura 2.17: Factorización LDL^T con pivotamiento basado en la inversa correspondiente al último nivel de la jerarquía que construye localmente la tarea hoja $(3,2)$.

Procesamiento paralelo de los separadores del segundo nivel

Tras la ejecución de las tareas hoja, el algoritmo paralelo procede con la ejecución de las tareas del segundo nivel, $(2,1)$ y $(2,2)$. Estas tareas realizan, de modo independiente, las computaciones de la segunda fase del Algoritmo 2.19, explotando el paralelismo que se pone de manifiesto en el cálculo de (2.96). Así, la tarea $(2,i)$, con $i=1,2$, factoriza $A_{jj}^{(2,i)}$, con $j=i+4$, según (2.98), aproxima los bloques correspondientes de A_Y e A_Y^T conforme a (2.99), y calcula sus contribuciones locales a $\hat{S}_{Z_C}$; ver (2.100). Con este propósito, la tarea $(2,i)$, debe obtener un sumando de (2.95), $A^{(2,i)}$, que contenga los bloques que ésta factoriza, junto con el bloque local correspondiente a su ancestro $(1,1)$. Estos sumandos se definen como

$$A = A^{(2,1)} + A^{(2,2)}$$

$$\left(\begin{array}{cc||c} A_{55}^{(2,1)} & 0 & (A_{75}^{(2,1)})^T \\ 0 & A_{66}^{(2,2)} & (A_{76}^{(2,2)})^T \\ \hline\hline A_{75}^{(2,1)} & A_{76}^{(2,2)} & A_{77}^{(1,1)} \end{array}\right) = \left(\begin{array}{cc||c} A_{55}^{(2,1)} & 0 & (A_{75}^{(2,1)})^T \\ 0 & 0 & 0 \\ \hline\hline A_{75}^{(2,1)} & 0 & A_{77}^{(2,1)} \end{array}\right) + \left(\begin{array}{cc||c} 0 & 0 & 0 \\ 0 & A_{66}^{(2,2)} & (A_{76}^{(2,2)})^T \\ \hline\hline 0 & A_{76}^{(2,2)} & A_{77}^{(2,2)} \end{array}\right),$$

donde $A_{77}^{(2,i)}$ es el bloque local de $(2,i)$ correspondiente a su ancestro $(1,1)$. Si reparticionamos los bloques que se encarga de factorizar la tarea $(2,1)$ teniendo en cuenta la definición de A en (2.95)

$$A_{55}^{(2,1)} = \begin{pmatrix} \hat{S}_{C,11}^{(3,1)} & 0 & (\hat{S}_{C,51}^{(3,1)})^T \\ 0 & \hat{S}_{C,22}^{(3,2)} & (\hat{S}_{C,52}^{(3,2)})^T \\ \hat{S}_{C,51}^{(3,1)} & \hat{S}_{C,52}^{(3,2)} & \mathbf{\hat{S}_{C,55}^{(2,1)}} \end{pmatrix}, \; A_{75}^{(2,1)} = \begin{pmatrix} \hat{S}_{C,71}^{(3,1)} & \hat{S}_{C,72}^{(3,2)} & \mathbf{\hat{S}_{C,75}^{(2,1)}} \end{pmatrix},$$

y hacemos lo propio con la tarea $(2,2)$

$$A_{66}^{(2,2)} = \begin{pmatrix} \hat{S}_{C,33}^{(3,3)} & 0 & (\hat{S}_{C,63}^{(3,4)})^T \\ 0 & \hat{S}_{C,44}^{(3,3)} & (\hat{S}_{C,64}^{(3,4)})^T \\ \hat{S}_{C,63}^{(3,3)} & \hat{S}_{C,64}^{(3,4)} & \mathbf{\hat{S}_{C,66}^{(2,2)}} \end{pmatrix}, \; A_{76}^{(2,2)} = \begin{pmatrix} \hat{S}_{C,73}^{(3,3)} & \hat{S}_{C,74}^{(3,4)} & \mathbf{\hat{S}_{C,76}^{(2,2)}} \end{pmatrix},$$

entonces podemos observar, por una parte, que los bloques que no están marcados en negrita se pueden obtener *directamente* de los complementos de Schur calculados por sus hijos en el paso 4 y, por otra, que los bloques marcados en negrita pertenecen a $\hat{S}_{C_Z}$; ver (2.92). Por lo tanto, para recuperar las entradas de estos últimos, la tarea $(2,1)$ debe acumular los bloques locales de contribución calculados por sus hijos conforme a (2.109),

$$\hat{S}^{(2,1)}_{C,55} = \hat{S}^{(3,1)}_{C,55} + \hat{S}^{(3,2)}_{C,55},\ \hat{S}^{(2,1)}_{C,75} = \hat{S}^{(3,1)}_{C,75} + \hat{S}^{(3,2)}_{C,75},$$

y la tarea $(2,2)$ hace lo propio con los de $(3,3)$ y $(3,4)$,

$$\hat{S}^{(2,2)}_{C,66} = \hat{S}^{(3,3)}_{C,66} + \hat{S}^{(3,4)}_{C,66},\ \hat{S}^{(2,2)}_{C,76} = \hat{S}^{(3,3)}_{C,76} + \hat{S}^{(3,4)}_{C,76}.$$

Por último, teniendo en cuenta que (ver (2.95))

$$A^{(1,1)}_{77} = \hat{S}^{(1,1)}_{C,77},$$

también pertenece a $\hat{S}_{C_Z}$ (ver (2.92)), y que, por tanto,

$$\hat{S}^{(1,1)}_{C,77} = \hat{S}^{(3,1)}_{C,77} + \hat{S}^{(3,2)}_{C,77} + \hat{S}^{(3,3)}_{C,77} + \hat{S}^{(3,4)}_{C,77},$$

entonces $(2,1)$ y $(2,2)$ obtienen $A^{(2,1)}_{77}$ y $A^{(2,2)}_{77}$ sumando los bloques locales de contribución de sus hijos como

$$A^{(2,1)}_{77} = \hat{S}^{(3,1)}_{C,77} + \hat{S}^{(3,2)}_{C,77}$$

y

$$A^{(2,2)}_{77} = \hat{S}^{(3,3)}_{C,77} + \hat{S}^{(3,4)}_{C,77}.$$

La Figura 2.18 ilustra gráficamente cómo la tarea $(2,1)$ construye su matriz de entrada, $A^{(2,1)}$, a partir de los complementos de Schur locales calculados por sus hijos en el paso 4. Para un caso con $l = 4$ niveles en el árbol de tareas, la Figura 2.19 muestra la descomposición de la matriz de la Figura 2.13 (b) (c) entre las tareas del tercer nivel. Los bloques de color rojo, azul, verde y magenta son los bloques que construyen, respectivamente, las tareas $(3,1)$, $(3,2)$, $(3,3)$ y $(3,4)$. Dentro de los bloques de un mismo color, los bloques con patrón rayado se obtienen *directamente* a partir de los complementos de Schur calculados por los hijos de la tarea y, los bloques con patrón denso y cruzado, acumulando sus bloques locales de contribución. Además, los bloques con patrón cruzado contienen las contribuciones locales de las tareas del tercer nivel a los bloques de color negro.

Tras construir $A^{(2,i)}$, las tareas $(2,i)$, con $i = 1, 2$, paralelizan las computaciones de la segunda fase del Algoritmo 2.19, aplicando localmente el siguiente algoritmo:

- En los pasos 1 y 2, la matriz de escalado diagonal, $D^{(2,i)}$, $A^{(2,i)} \rightarrow D^{(2,i)}A^{(2,i)}D^{(2,i)} = \tilde{A}^{(2,i)}$, y la permutación para reducción de llenado, $\tilde{P}^{(2,i)}$, $\tilde{A}^{(2,i)} \rightarrow (\tilde{P}^{(3,i)})^T \tilde{A}^{(2,i)} \tilde{P}^{(2,i)} = \hat{A}^{(2,i)}$, se restringen a los bloques que factoriza la tarea. Por lo tanto, si particionamos $D^{(2,i)}$ y $\tilde{P}^{(2,i)}$ conforme a $A^{(2,i)}$, entonces

$$D^{(2,i)} = \left(\begin{array}{c|c} D^{(2,i)}_{jj} & 0 \\ \hline 0 & I \end{array}\right) \text{ y } \tilde{P}^{(2,i)} = \left(\begin{array}{c|c} \tilde{P}^{(2,i)}_{jj} & 0 \\ \hline 0 & I \end{array}\right).$$

- En el paso 3, la tarea $(2,i)$ aplica el núcleo de factorización LDL^T con pivotamiento basado en la inversa a la matriz $\hat{A}^{(2,i)}$ de forma que la estrategia de pivotamiento no considera elementos pivote del bloque local correspondiente a su ancestro y, por tanto,

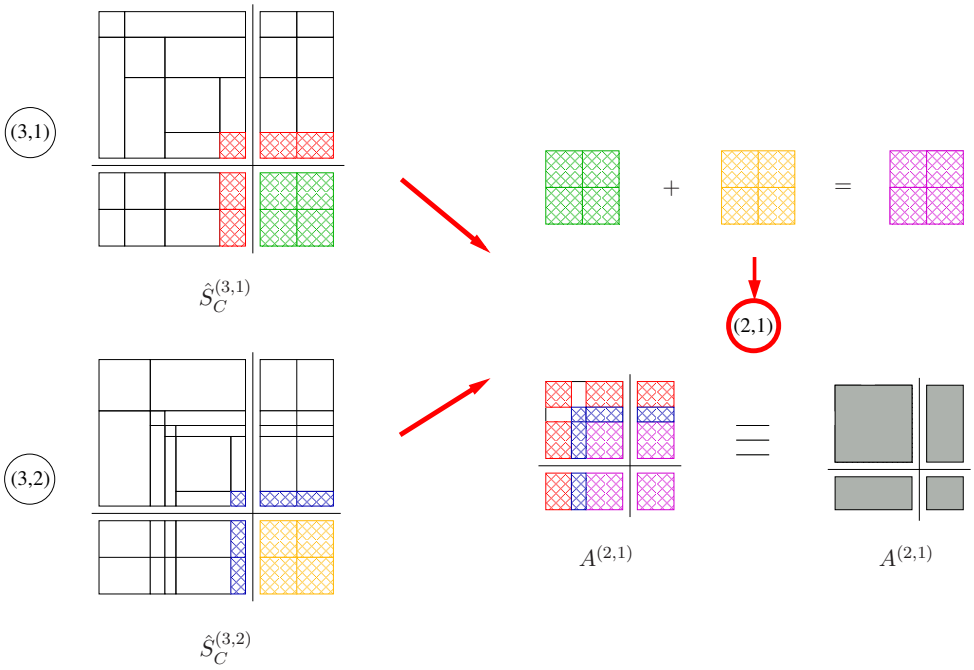


Figura 2.18: La tarea $(2,1)$ construye su matriz de entrada, $A^{(2,1)}$, a partir de los complementos de Schur resultantes de las computaciones efectuadas localmente por sus hijos, $(3,1)$ y $(3,2)$.

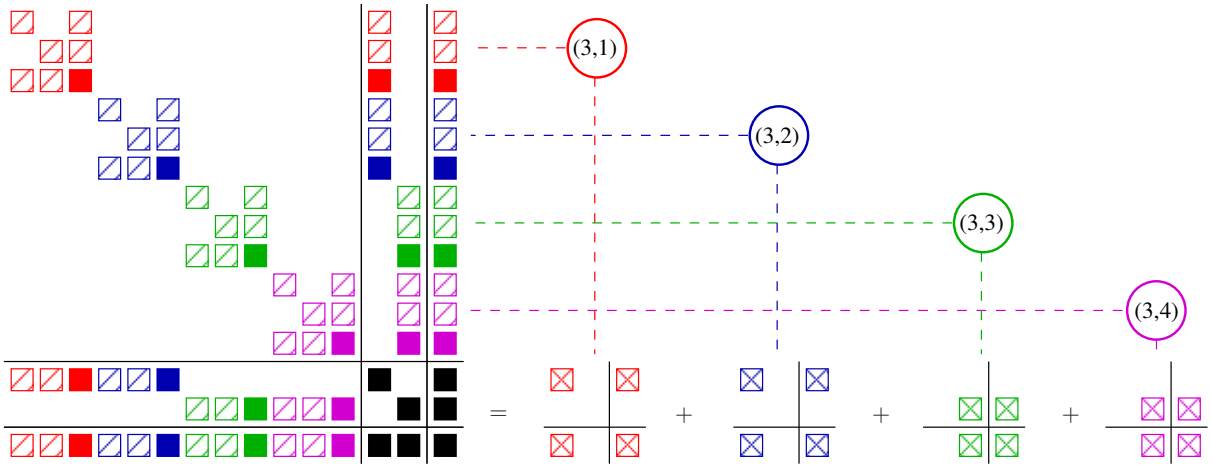


Figura 2.19: Descomposición de la matriz de entrada, $A \equiv \Phi^T \hat{S}_C \Phi$, para la segunda fase del Algoritmo 2.19 entre las tareas del tercer nivel del árbol de tareas; ver Figura 2.13 (b) (c).

la factorización incompleta por bloques que se obtiene como resultado tiene la siguiente estructura

$$(P^{(2,i)})^T \hat{A}^{(2,i)} P^{(2,i)} = \left(\begin{array}{cc|c} B_{jj}^{(2,i)} & (F_{jj}^{(2,i)})^T & (F_{7j}^{(2,i)})^T \\ F_{jj}^{(2,i)} & C_{jj}^{(2,i)} & (C_{7j}^{(2,i)})^T \\ \hline F_{7j}^{(2,i)} & C_{7j}^{(2,i)} & C_{77}^{(2,i)} \end{array}\right) \approx \tilde{L}^{(2,i)} \tilde{D}^{(2,i)} (\tilde{L}^{(2,i)})^T, \quad (2.115)$$

con

$$\tilde{L}^{(2,i)} = \left(\begin{array}{cc|c} \tilde{L}_{B,jj}^{(2,i)} & 0 & 0 \\ \tilde{L}_{F,jj}^{(2,i)} & I & 0 \\ \hline \tilde{L}_{F,7j}^{(2,i)} & 0 & I \end{array}\right), \tilde{D}^{(2,i)} = \left(\begin{array}{cc|c} \tilde{D}_{B,jj}^{(2,i)} & 0 & 0 \\ 0 & \hat{S}_{C,jj}^{(2,i)} & (\hat{S}_{C,7j}^{(2,i)})^T \\ \hline 0 & \hat{S}_{C,7j}^{(2,i)} & \hat{S}_{C,77}^{(2,i)} \end{array}\right).$$

Los bloques $B_{jj}^{(2,i)}$ y $C_{jj}^{(3,i)}$ contienen, respectivamente, los elementos pivote aceptados y rechazados localmente por la tarea $(2,i)$. $C_{77}^{(2,i)}$ es el bloque local de $(3,i)$ correspondiente a su ancestro; por último, $\hat{S}_{C,77}^{(2,i)}$ es el bloque local de contribución de $(2,i)$ a $Z_{\hat{S}_C}$ en la ecuación (2.100).

- En el paso 4, la tarea $(2,i)$ procede recursivamente con el siguiente nivel, repitiendo los pasos 1, 2 y 3 con el complemento de Schur aproximado como matriz de entrada al siguiente nivel; es decir, con

$$A^{(2,i)} \equiv \left(\begin{array}{c|c} \hat{S}_{C,jj}^{(2,i)} & (\hat{S}_{C,j7}^{(2,i)})^T \\ \hline \hat{S}_{C,7j}^{(2,i)} & \hat{S}_{C,77}^{(2,i)} \end{array}\right), \quad (2.116)$$

hasta que el tamaño del bloque $\hat{S}_{C,jj}^{(2,i)}$ es lo "suficientemente pequeño". Finalmente, $(2,i)$ "envía" (2.116) a $(1,1)$.

Procesamiento del separador del primer nivel

La tarea raíz, $(1,1)$, es responsable de las computaciones que se realizan en la tercera y última fase del Algoritmo 2.19. En la matriz de entrada a esta fase, ver (2.101), el bloque $\hat{S}_{C,77}^{(1,1)}$ pertenece a $\hat{S}_{C_Z}$ y, por tanto, la tarea $(1,1)$ recupera este bloque acumulando los bloques de contribución de $(2,1)$ y $(2,2)$ como

$$\hat{S}_{C,77}^{(1,1)} = \hat{S}_{C,77}^{(2,1)} + \hat{S}_{C,77}^{(2,2)}.$$

El resto de bloques de (2.101) se obtienen *directamente* a partir de (2.116), con $i = 1, 2$. Cabe destacar que el código que efectúa las computaciones de la tarea raíz explota un tipo diferente de paralelismo, si se invoca una versión multihebra de la biblioteca BLAS al factorizar el sistema complemento de Schur mediante una factorización LDL^T densa; ver paso 6 del Algoritmo 2.19.

Adaptación de las reglas de descarte y de la técnica de Ajiz-Jennings al caso paralelo

En cada uno de los pasos de eliminación, el núcleo de factorización LDL^T con pivotamiento basado en la inversa descarta elementos situados en la parte triangular inferior estricta de la columna actual del factor $\tilde{L}$, si estos elementos cumplen las reglas de descarte (2.70) y (2.71). Con este propósito, este núcleo no forma explícitamente el complemento de Schur aproximado correspondiente a cada paso de eliminación sino que, en su lugar, calcula los elementos de la columna actual de esta matriz, junto con los elementos de la diagonal principal. Estos últimos los necesita para las reglas de descarte (2.70) y (2.71), así como para la técnica de Ajiz-Jennings que, por cada elemento descartado, $\hat{s}_{mj}/\hat{s}_{jj}$, compensa los elementos $\hat{s}_{jj}$ y $\hat{s}_{mm}$ de la diagonal

principal del complemento de Schur aproximado correspondiente al paso de eliminación actual, sumándoles los términos

$$\sqrt{\frac{\hat{s}_{jj}}{\hat{s}_{mm}}}|\hat{s}_{mj}| \quad \text{y} \quad \sqrt{\frac{\hat{s}_{mm}}{\hat{s}_{jj}}}|\hat{s}_{mj}|,$$

respectivamente. Ambos mecanismos se fundamentan en propiedades que sólo cumplen las matrices SDP y, por tanto, para que operen apropiadamente, es necesario que el complemento de Schur aproximado correspondiente a cada paso de eliminación sea SDP. Esto puede causar problemas en el caso paralelo, ya que, aunque la matriz A es SDP, existe la posibilidad de que no lo sea la matriz $A^{(3,i)}$ que procesa localmente $(3,i)$ (ver Figura 2.15) y, por tanto, tampoco hay garantías de que lo sea el complemento de Schur aproximado correspondiente en cada paso de la factorización. En este apartado discutimos este problema en detalle y presentamos cómo adaptar las reglas de descarte y la técnica de Ajiz-Jennings para gestionarlo.

Para el desarrollo siguiente, asumimos que el sumando correspondiente a la tarea hoja del árbol $(3,2)$, $A^{(3,2)} \in \mathbb{R}^{t,t}$, se particiona como

$$A^{(3,2)} = \left(\begin{array}{c|c|c} A_{22}^{(3,2)} & (A_{52}^{(3,2)})^T & (A_{72}^{(3,2)})^T \\ \hline A_{52}^{(3,2)} & A_{55}^{(3,2)} & (A_{75}^{(3,2)})^T \\ \hline A_{72}^{(3,2)} & A_{75}^{(3,2)} & A_{77}^{(3,2)} \end{array}\right),$$

donde $A_{22} \in \mathbb{R}^{t_1,t_1}$, $A_{55}^{(3,2)} \in \mathbb{R}^{t_2,t_2}$ y $A_{77}^{(3,2)} \in \mathbb{R}^{t_3,t_3}$. Teniendo en cuenta la descomposición inicial de la matriz de entrada para la primera fase del Algoritmo 2.19 (ver Figura 2.15) los bloques locales de la tarea $(3,2)$ correspondientes al ancestro $(2,1)$ se inicializan como

$$A_{55}^{(3,2)} = \frac{1}{2}A_{55}^{(2,1)},\ A_{75}^{(3,2)} = \frac{1}{2}A_{75}^{(2,1)},\ (A_{75}^{(3,2)})^T = \frac{1}{2}(A_{75}^{(2,1)})^T,$$

y el bloque local correspondiente al ancestro $(1,1)$, como

$$A_{77}^{(3,2)} = \frac{1}{4}A_{77}^{(1,1)}.$$

Por lo tanto, $A^{(3,2)}$ tiene como elementos de la diagonal principal

$$\left(\begin{array}{ccc|ccc|ccc} a_{11} & \dots & a_{t_1,t_1} & \frac{1}{2}a_{t_1+1,t_1+1} & \dots & \frac{1}{2}a_{t_1+t_2,t_1+t_2} & \frac{1}{4}a_{t_1+t_2+1,t_1+t_2+1} & \dots & \frac{1}{4}a_{t,t} \end{array}\right), \tag{2.117}$$

donde $a_{11},\dots,a_{t_1,t_1}$; $a_{t_1+1,t_1+1},\dots,a_{t_1+t_2,t_1+t_2}$; y $a_{t_1+t_2+1,t_1+t_2+1},\dots,a_{t,t}$; son, respectivamente, los elementos de la diagonal principal de $A_{55}^{(3,2)} = A_{55}^{(3,1)}$; $A_{55}^{(2,1)}$; y $A_{77}^{(1,1)}$.

Por simplicidad, asumimos que, tras los primeros $1 < (j-1) < t_1$ pasos de eliminación, la factorización LDL^T con pivotamiento basado en la inversa que emplea localmente la tarea $(3,2)$ ha aceptado los primeros $j-1$ elementos pivote de $A^{(3,2)}$ y que, por tanto, el complemento de Schur aproximado correspondiente al paso de eliminación actual se puede particionar de la siguiente forma

$$\left(\begin{array}{cc|ccc|ccc} \hat{s}_{jj} & \cdots & \cdots & \hat{s}_{kj} & \cdots & \cdots & \hat{s}_{lj} & \cdots & \hat{s}_{jt} \\ \vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \hline \vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \hat{s}_{kj} & \cdots & \cdots & \hat{s}_{kk} & \cdots & \cdots & \hat{s}_{lk} & \cdots & \hat{s}_{tk} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \hline \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \hat{s}_{lj} & \cdots & \cdots & \hat{s}_{lk} & \cdots & \cdots & \hat{s}_{ll} & \cdots & \hat{s}_{tl} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots & \ddots & \vdots \\ \hat{s}_{tj} & \cdots & \cdots & \hat{s}_{tk} & \cdots & \cdots & \hat{s}_{tl} & \cdots & \hat{s}_{tt} \end{array}\right),$$

donde $t_1 + 1 \leq k \leq t_1 + t_2$; $t_1 + t_2 + 1 \leq l \leq t$. Además, asumimos que, en el paso j-ésimo, la factorización acepta el elemento pivote $\hat{s}_{jj}$. Por tanto, para generar una nueva fila del factor aproximado $\tilde{L}^{(3,2)}$, se aplican las reglas de descarte (2.70) y (2.71) a cada elemento $\hat{s}_{mj}/\hat{s}_{jj}$, con $m = j + 1, \ldots, t$.

Si consideramos el descarte de los elementos $\hat{s}_{kj}/\hat{s}_{jj}$ y $\hat{s}_{lj}/\hat{s}_{jj}$, y tenemos en cuenta (2.117) y que se han eliminado las primeras $j-1$ filas/columnas de $A^{(3,2)}$, entonces las entradas de la diagonal principal involucradas en el descarte de $\hat{s}_{kj}$ y $\hat{s}_{lj}$, se definen como

$$\hat{s}_{jj} = a_{jj} - \sum_{i=1}^{j-1} \tilde{l}_{ij}^{\,2} + r_{jj}, \tag{2.118}$$

$$\hat{s}_{kk} = \frac{1}{2} a_{kk} - \sum_{i=1}^{j-1} \tilde{l}_{ik}^{\,2} + r_{kk}, \tag{2.119}$$

$$\hat{s}_{ll} = \frac{1}{4} a_{ll} - \sum_{i=1}^{j-1} \tilde{l}_{il}^{\,2} + r_{ll}, \tag{2.120}$$

donde r_{jj}, r_{kk} y r_{ll} se refieren, respectivamente, a la suma de todas las compensaciones que la técnica de Ajiz-Jennings ha efectuado por cada elemento descartado de la columna j-ésima, k-ésima y l-ésima de $\tilde{L}^{(3,2)}$ en las primeras $j-1$ iteraciones. *La definición de $\hat{s}_{kk}$ y $\hat{s}_{ll}$ no garantiza que estas entradas pertenezcan a una matriz SDP*, dado que los términos $\frac{1}{2}a_{kk}$ y $\frac{1}{4}a_{ll}$ reducen el peso de la diagonal principal y pueden ser demasiado pequeños en relación a las actualizaciones resultantes de la eliminación de las primeras $j-1$ columnas. De hecho, (2.119) y (2.120) pueden tomar valores negativos, causando que el núcleo de factorización falle al calcular la compensación de Ajiz-Jennings correspondiente.

Una forma natural de resolver este problema es considerar, en lugar de $\hat{s}_{kk}$ y $\hat{s}_{ll}$, las entradas correspondientes del complemento de Schur aproximado que se hubiese obtenido eliminando las primeras $j-1$ filas y columnas de la matriz

$$\left(\begin{array}{c|c|c} A_{22}^{(3,2)} & (A_{52}^{(3,2)})^T & (A_{72}^{(3,2)})^T \\ \hline A_{52}^{(3,2)} & A_{55}^{(2,1)} & (A_{75}^{(2,1)})^T \\ \hline A_{72}^{(3,2)} & A_{75}^{(2,1)} & A_{77}^{(1,1)} \end{array}\right), \tag{2.121}$$

que sí es SDP. Estudiando la relación que existe entre la eliminación de esta matriz y la de $A^{(3,2)}$, se puede comprobar que la entrada k-ésima de la diagonal principal del complemento de Schur aproximado que nos interesa se obtiene *de forma económica* como

$$\frac{1}{2} a_{kk} + \hat{s}_{kk}, \quad \text{con } k = t_1 + 1, \ldots, t_1 + t_2,$$

y la situada en la entrada l-ésima, calculando

$$\frac{3}{4} a_{ll} + \hat{s}_{ll}, \quad \text{con } l = t_1 + t_2 + 1, \ldots, t.$$

Para implementar esta idea, la tarea $(3,2)$ emplea un vector $\hat{a}$ que contiene el valor cero en sus primeras t_1 posiciones, el valor $\frac{1}{2}a_{kk}$, con $k = t_1 + 1, \ldots, t_1 + t_2$, para las t_2 posiciones siguientes, y el valor $\frac{3}{4}a_{ll}$, con $l = t_1 + t_2 + 1, \ldots, t$, para las últimas t_3 posiciones. Nos referiremos a $\hat{a}$ como *vector subsidiario de la tarea*. Este vector se inicializa al mismo tiempo que las entradas de $A^{(3,2)}$ y la tarea $(3,2)$ no modifica su valor. Posteriormente, el núcleo de factorización LDL^T con pivotamiento basado en la inversa descarta un elemento $\hat{s}_{mj}/\hat{s}_{jj}$ si se cumple que

$$\left| \frac{\hat{s}_{mj}}{\hat{s}_{jj} + \hat{a}_j} \right| \max_{j \leq l \leq t} |\hat{s}_{lj}| \leq \frac{\tau}{\|e_j^T (\tilde{L}_j^{(3,2)})^{-1}\|_\infty} |\hat{s}_{mm} + \hat{a}_m|,$$

y

$$\left|\frac{\hat{s}_{mj}}{\hat{s}_{jj}+\hat{a}_j}\right| \max_{j+1\le l\le t}\left|\frac{\hat{s}_{lj}}{\hat{s}_{ll}+\hat{a}_l}\right| \le \frac{\tau}{\|e_j^T(\tilde{L}_j^{(3,2)})^{-1}\|_\infty};$$

cuando este elemento se descarta, la técnica de Ajiz-Jennings compensa $\hat{s}_{jj}$ y $\hat{s}_{mm}$, sumándoles

$$\sqrt{\frac{\hat{s}_{jj}+\hat{a}_j}{\hat{s}_{mm}+\hat{a}_m}}|\hat{s}_{mj}| \quad \text{y} \quad \sqrt{\frac{\hat{s}_{mm}+\hat{a}_m}{\hat{s}_{jj}+\hat{a}_j}}|\hat{s}_{mj}|,$$

respectivamente. Como se puede observar en estas expresiones, cada vez que se referencia a un elemento de la diagonal principal del complemento de Schur aproximado, se suma el elemento correspondiente del vector subsidiario. De esta forma se obtienen los elementos correspondientes del complemento de Schur aproximado si se hubiese factorizado (2.121).

La misma idea se puede aplicar a las tareas del resto de niveles del árbol, aunque en este caso las entradas del vector subsidiario se inicializan tomando como base el vector subsidiario *de cualquiera de sus hijos*. Sin embargo, dado que las tareas con hijos acumulan los bloques locales de contribución de sus hijos, es necesario que las primeras reajusten el vector subsidiario para que refleje el nuevo estado de las entradas de la diagonal principal de sus bloques locales de contribución. Por ejemplo, en el caso de la tarea $(2,1)$, las entradas del vector subsidiario correspondiente a su bloque local de contribución se obtienen reajustando las últimas t_3 posiciones del vector subsidiario de $(3,1)$ (o de $(3,2)$) como

$$\frac{3}{4}a_{ll} \to \frac{1}{2}a_{ll}, \quad \text{con } l = t_1+t_2+1,\ldots,t.$$

2.3.4. Diseño y justificación de las estrategias de mapeado y planificación

El siguiente paso en el proceso de diseño del algoritmo paralelo es escoger las estrategias que se encargan de *asignar* (mapear) las tareas en las que se descompone la construcción del precondicionador a los *procesos* disponibles, y *planificar* su ejecución concurrente conforme a las dependencias entre tareas capturadas en el árbol de dependencias. En el resto de este documento usaremos el término *proceso* para referirnos a una entidad abstracta que *ejecuta tareas*, es decir, que es capaz de usar el código y los datos de cualquier tarea para producir los datos de salida correspondientes a las computaciones que ésta efectúa. Asimismo, consideraremos que existe una relación uno a uno entre procesos y procesadores físicos del multiprocesador de memoria compartida, y clarificamos que, *en este contexto*, un proceso equivale a una hebra o hilo de ejecución del sistema en tiempo de ejecución.

En el diseño del algoritmo paralelo, la fase inicial de particionado y las estrategias de mapeado y planificación de tareas se complementan para *equilibrar la carga en la computación de las tareas hoja*. La experiencia adquirida con el algoritmo paralelo revela que gran parte del coste computacional y de almacenamiento requerido por el método numérico paralelo está concentrado en las tareas hoja del árbol y, por lo tanto, el equilibrio de carga en la computación de las tareas hoja tiene un impacto *significativo* sobre la eficiencia de nuestro algoritmo paralelo. En este apartado describimos las soluciones que incorporamos en la fase de particionado y las estrategias de mapeado y planificación de tareas, que se diseñan para equilibrar la carga computacional de las tareas hoja entre los distintos procesos.

El objetivo de la fase inicial de particionado es separar el grafo de adyacencia de la matriz del sistema en subgrafos independientes con (aproximadamente) el mismo número de vértices, de forma que el número de vértices que los separe sea lo más pequeño posible. En el contexto de nuestro algoritmo paralelo, el primer requisito que debe cumplir el precondicionado es equilibrar la carga computacional de las tareas hoja y, el segundo, minimizar las demandas computacionales de las tareas que se encargan de las computaciones de la segunda fase del Algoritmo 2.19

en adelante. Este segundo requisito es de vital importancia para la eficiencia del algoritmo paralelo, ya que el grado de concurrencia que se puede explotar en el árbol de tareas se reduce paulatinamente a medida que se van procesando los niveles más cercanos a la raíz.

El estado del arte en materia de algoritmos de particionado de grafos incluye una familia de heurísticos para el cálculo de separadores de vértices, denominados heurísticos multinivel [89, 102], a través de los cuales la disección anidada (*MultiLevel-based Nested Dissection*) obtiene, con un *coste computacional relativamente moderado*, particionados cercanos al óptimo en el contexto de muchos problemas, como por ejemplo, en aquellos problemas 3D que emplean mallas de elementos o diferencias finitas para discretizar el dominio de solución de la EDP. Además, las mallas que surgen en el contexto de este último tipo de aplicaciones tienen, con frecuencia, la favorable propiedad de que el tamaño de los separadores es varios órdenes de magnitud más pequeño que el tamaño de los subgrafos independientes. Por ejemplo, el tamaño de los separadores es de orden $O(n^{1/2})$ para mallas regulares en 2D, y de orden $O(n^{2/3})$ para mallas regulares en 3D, donde n es el número de nodos de la malla [112].

A pesar de los factores favorables mencionados en el párrafo anterior, *la distribución final del coste computacional del cálculo del precondicionador entre las tareas del árbol depende de factores que no se pueden cuantificar a priori* como, por ejemplo, la distribución y número de nodos aceptados y rechazados en cada tarea, o de cómo se propaga el llenado durante la construcción del precondicionador que calcula Algoritmo 2.19. La experiencia adquirida con el algoritmo paralelo nos ha permitido observar que:

- El llenado en $\hat{S}_C$ no crece de forma considerable para las fases en las que se puede explotar un bajo grado de concurrencia (ver, por ejemplo, los Algoritmos 2.17 y 2.18). Si se diese el caso, entonces la eficiencia del algoritmo paralelo se reduciría significativamente.

  La justificación de esta observación experimental se puede esbozar teniendo en cuenta cuál es el objetivo de las técnicas de precondicionado multinivel de ILUPACK, a saber, *mejorar* la escalabilidad de los métodos iterativos basados en la generación de subespacios de Krylov cuando se aplican a problemas de gran dimensión. Esta mejora se obtiene logrando un compromiso *razonable* entre la eficacia del precondicionador para reducir el número de condición del sistema precondicionado, y el coste computacional y de almacenamiento requerido para construirlo y aplicarlo. En el contexto de las técnicas de precondicionado basadas en factorizaciones ILU multinivel, una condición necesaria para desarrollar resolutores (precondicionados) escalables es controlar eficientemente el llenado que se produce durante el proceso de construcción del precondicionador. Con este propósito, el Algoritmo 2.19 aplica reglas de descarte a las entradas del factor $\tilde{L}$ durante la factorización del paso 3, a las entradas de $\hat{S}_C$ entre los pasos 3 y 4, y emplea heurísticos de reducción de llenado en el paso 2. Además, en el trabajo desarrollado en [31], se demostró, en el contexto de sistemas de ecuaciones que surgen en la discretización de EDPs elípticas, que las propiedades del complemento de Schur aproximado, $\hat{S}_C$, resultante de aplicar la estrategia de pivotamiento basado en la inversa, permiten a las estrategias de descarte de elementos reducir eficazmente las demandas computacionales y de almacenamiento necesarias para las computaciones de niveles subsecuentes. En el algoritmo paralelo, las tareas que paralelizan las computaciones de la segunda fase en adelante, calculan la factorización multinivel del complemento de Schur aproximado resultante tras aplicar recursivamente la factorización LDL^T con pivotamiento basado en la inversa a los subgrafos independientes. En consecuencia, se puede afirmar, al menos en aplicaciones que surgen en la discretización de EDPs elípticas, que estas propiedades favorables del algoritmo numérico permiten controlar razonablemente las demandas computacionales de las tareas más cercanas a la raíz.

- Equilibrar el número de nodos de los subgrafos independientes no es siempre suficiente para equilibrar la carga computacional de las tareas hoja entre los distintos procesos (primero

de los requisitos de nuestro algoritmo paralelo).

Por ejemplo, la EDP de difusión en estado estacionario *con coeficientes discontinuos* es una variante de la EDP básica en la que los coeficientes de las derivadas parciales varían en función de la región del dominio computacional. Teniendo en cuenta que el (coste computacional del) método de ILUPACK es sensible al valor de estos coeficientes (ver Apartado 2.2.4), esta propiedad del problema puede ocasionar desequilibrios de carga severos en el algoritmo paralelo, incluso cuando las tareas hoja del árbol tienen aproximadamente el mismo número de nodos; ver Apartado 2.3.5. Esta observación motiva el diseño y desarrollo de técnicas para la fase de particionado y las estrategias de mapeado y planificación de tareas que gestionen eficazmente la posible irregularidad del coste computacional de las tareas hoja.

En la fase inicial de particionado, la disección anidada particiona recursivamente aquellas tareas hoja que concentran, *potencialmente*, un mayor coste computacional. Este coste depende de factores que no se pueden cuantificar, al menos de forma precisa, antes de la ejecución del algoritmo numérico. La fase de particionado emplea, en su lugar, el número de aristas del subgrafo independiente correspondiente a la tarea, $G_{(i,j)} = \{V_{(i,j)}, E_{(i,j)}\}$, es decir, $|E_{(i,j)}|$, para determinar si éste concentra, *potencialmente*, un coste computacional relativamente alto. En concreto, la disección anidada particiona $G_{(i,j)}$ si se cumple que

$$\frac{|E_{(i,j)}|}{|E|} > \frac{1}{f}, \tag{2.122}$$

donde $f \geq 1$ es un parámetro preestablecido que actúa como umbral superior del número de aristas de los subgrafos independientes y $|E|$ es el número de aristas de $G(A)$. En aquellos grafos en los que el grado (número de aristas) de cada vértice es constante, este criterio equivale a

$$\frac{|V_{(i,j)}|}{|V|} > \frac{1}{f}. \tag{2.123}$$

Sin embargo, para grafos que no cumplen esta propiedad, y en los que existen variaciones significativas entre los grados de sus vértices, el criterio (2.122) gestiona el potencial desequilibrio causado por la variabilidad en el número de aristas particionando aquellos subgrafos independientes que concentran un elevado número de aristas.

La fase de particionado toma como entrada un determinado subgrafo independiente, $G_{(i,j)}$, el grafo de adyacencia de la matriz del sistema, $G(A)$, y el parámetro f; ver Algoritmo 2.20. El algoritmo particiona $G_{(i,j)}$ recursivamente hasta que, finalmente, ningún subgrafo independiente del subárbol de tareas correspondiente, $T_{(i,j)}$, cumple el criterio (2.122). Durante el proceso de particionado, el algoritmo también numera los vértices de $G_{(i,j)}$, obteniendo la permutación correspondiente, $\Pi_{(i,j)}$. Cuando $G_{(i,j)}$ cumple (2.122), el Algoritmo 2.20 invoca (línea 2) un heurístico para el cálculo de separadores de vértices, obteniendo el separador $S_{(i,j)}$, y dos subgrafos independientes *nuevos*, $G_{(i+1,2j-1)}$ y $G_{(i+1,2j)}$. Posteriormente, el subárbol de tareas y la permutación asociada a $G_{(i+1,2j-1)}$ y $G_{(i+1,2j)}$ se obtienen, respectivamente, mediante disección anidada (líneas 3 y 4). Por último, el subárbol de tareas y la permutación asociada a $G_{(i,j)}$ se obtienen combinando (líneas 6 y 7), respectivamente, los resultados obtenidos en las líneas 3, 4 y 5. Cuando $G_{(i,j)}$ no cumple (2.122), se obtiene una nueva hoja del árbol de tareas (línea 9), y sus vértices se numeran (línea 10) utilizando, por ejemplo, una estrategia de reordenación para reducción de llenado como AMD, RCM o MLND (ver Apartado 2.1.3). Variando el valor del parámetro f, se puede ajustar el grado de concurrencia disponible para la fase de cálculo del precondicionador: al aumentar el valor de f, se obtienen árboles de tareas más profundos y/o con mayor número de hojas. La experiencia adquirida con el algoritmo paralelo indica que valores de f en el invervalo $[p, 3p]$, con p el número de procesos involucrados en el cómputo paralelo, son apropiados para muchos ejemplos numéricos, aunque el óptimo depende en gran medida de cada

problema concreto. *Esta elección del valor de f produce, con frecuencia, árboles de tareas con más tareas hoja que procesos, aumentando la probabilidad de mejorar el equilibrio de la carga computacional.*

Algoritmo 2.20: Disección anidada.

Entrada: $\{G_{(i,j)} = \{V_{(i,j)}, E_{(i,j)}\}, G(A) = \{V, E\}, f\}$
Salida : $\{T_{(i,j)}, \Pi_{(i,j)}\}$

1 **si** $\frac{|E_{(i,j)}|}{|E|} > \frac{1}{f}$ **entonces**
2 $\quad\{S_{(i,j)}, G_{(i+1,2j-1)}, G_{(i+1,2j)}\} \leftarrow$ calcular_separador $(G_{(i,j)})$
3 $\quad\{T_{(i+1,2j-1)}, \Pi_{(i+1,2j-1)}\} \leftarrow$ disección_anidada $(G_{(i+1,2j-1)}, G(A), f)$
4 $\quad\{T_{(i+1,2j)}, \Pi_{(i+1,2j)}\} \leftarrow$ disección_anidada $(G_{(i+1,2j)}, G(A), f)$
5 $\quad\Pi_{(i,j)} \leftarrow$ ordenar $(S_{(i,j)})$
6 $\quad\Pi_{(i,j)} \leftarrow [\ \Pi_{(i+1,2j-1)}, \Pi_{(i+1,2j)}, \Pi_{(i,j)}\ ]$
7 $\quad T_{(i,j)} \leftarrow$ crear_árbol (raíz=(i,j), izq=$T_{(i+1,2j-1)}$, der=$T_{(i+1,2j)}$)
8 **si no**
9 $\quad T_{(i,j)} \leftarrow (i,j)$
10 $\quad\Pi_{(i,j)} \leftarrow$ ordenar $(G_{(i,j)})$
11 **fin si**

En este punto del desarrollo es necesario introducir el concepto de tarea *ejecutable* (o en estado ejecutable), y qué reglas emplea la *estrategia de planificación* que incorpora el algoritmo paralelo para decidir cuál de las tareas que se encuentran en dicho estado se va a ejecutar a continuación. Una tarea en estado ejecutable es aquella cuyas dependencias ya han sido resueltas en el estado actual de la computación paralela. Inicialmente, todas las tareas hoja del árbol son tareas ejecutables, mientras que las tareas que se encargan de procesar los separadores no son ejecutables. A medida que el algoritmo va completando los cálculos correspondientes, tareas inicialmente no ejecutables cambian su estado a ejecutables. Por ejemplo, en el caso del árbol de la Figura 2.14, el estado de la tarea $(2,1)$ cambia a ejecutable en el instante en el que se completa la ejecución de $(3,1)$ y $(3,2)$. Cuando un determinado proceso queda ocioso al completar la ejecución de una tarea, la estrategia de planificación tiene que decidir qué tarea ejecutará este proceso en siguiente lugar de entre las tareas que actualmente se encuentran en estado ejecutable. Para tomar esta decisión, la estrategia de planificación tiene en cuenta las siguientes reglas:

1. Las tareas hoja tienen prioridad frente a las tareas con hijos.

2. Entre las tareas hoja, tiene prioridad aquella que, *potencialmente*, concentre mayor coste computacional. Al igual que en el caso del criterio (2.122), se emplea el número de aristas del subgrafo independiente correspondiente a la tarea hoja para tomar esta decisión. El objetivo de esta regla es priorizar la ejecución de las tareas hoja más costosas y, de esa forma, reducir la posibilidad de que se produzca desequilibrio de carga debido a la ejecución tardía de éstas.

3. Entre las tareas con hijos, se escoge, en primer lugar, la tarea que lleve más tiempo en estado ejecutable.

El Algoritmo 2.21 implementa la estrategia de planificación de tareas basada en las reglas anteriores, y combina los bloques básicos presentados en el Apartado 2.3.3 para construir, en paralelo, precondicionadores basados en la factorización LDL^T multinivel. El algoritmo mantiene una cola centralizada, Q, que *sólo* contiene tareas ejecutables. Esta estructura compartida se inicializa (línea 2) con todas las hojas del árbol en orden decreciente con respecto al número

de aristas del subgrafo independiente correspondiente a cada hoja (consultar la segunda de las reglas de planificación). Como resultado de esta inicialización, la tarea hoja cuyo subgrafo independiente concentra un mayor número de aristas se sitúa en la cabeza de Q y, la de menor número, en su cola. Posteriormente, se crea un grupo de procesos (línea 4), que monitorizan Q a la espera de nuevas tareas ejecutables (bucle **repetir**), y entran en el bucle **mientras** siempre que queden tareas en Q por ejecutar. *La asignación de tareas a procesos (línea 9) es completamente dinámica, lo que persigue mejorar, en tiempo de ejecución, el equilibrio de la carga computacional.* La línea 10 construye el sumando correspondiente a una tarea conforme a la Figura 2.15, en el caso de las tareas hoja y, conforme a la Figura 2.18, en el caso de las tareas encargadas de procesar los separadores. El núcleo computacional del algoritmo se representa a través de la llamada a la rutina calcular_factorización (línea 11), que construye la factorización LDL^T multinivel del sumando correspondiente a la tarea; ver Figuras 2.16 y Figura 2.17. En este algoritmo paralelo no existen puntos de sincronización en el interior de la rutina calcular_factorización. Esta propiedad es una consecuencia de descomponer las matrices involucradas en las computaciones que efectúa el Algoritmo 2.19. Cuando un proceso completa la ejecución de una tarea, comprueba si se ha completado la ejecución de la tarea hermana y, en dicho caso, inserta la tarea padre de ambas en Q (línea 14). Por simplicidad, en el Algoritmo 2.21 se han omitido los detalles del acceso concurrente seguro a las estructuras de datos centralizadas.

Algoritmo 2.21: Cálculo paralelo de la factorización LDL^T multinivel.

```
1  [ Π , T ] ← disección_anidada(G(A), G(A), f)          ▷ ver Algoritmo 2.20
2  Q ← { hojas(T) }          ▷ inicializar Q con tareas hoja de T en orden decreciente por #aristas
3  marcar todas las tareas de T como no ejecutadas
4  Inicio región paralela                                 ▷ crear grupo de procesos
5      pid ← obtener_pid()                                ▷ obtener identificador de proceso
6      repetir
7          mientras Q no esté vacía hacer
8              tid ← desencolar(Q)                        ▷ extraer tarea ejecutable de la cabeza de Q
9              map [ tid ] ← pid                          ▷ proceso pid a cargo de la tarea tid
10             construir_sumando(tid)                     ▷ construir sumando correspondiente a tid
11             calcular_factorización(tid)    ▷ calcular factorización LDL^T multinivel correspondiente a tid
12             marcar tid como ejecutada
13             si se han resuelto las dependencias de padre(tid) entonces
14                 encolar(padre(tid), Q)                 ▷ insertar nueva tarea ejecutable en la cola de Q
15             fin si
16         fin mientras
17     hasta todas las tareas ejecutadas
18 Fin región paralela
```

Al planificar la ejecución de las tareas del árbol, el Algoritmo 2.21 no tiene en cuenta la "ubicación" de los datos de salida que éstas producen como criterio para mapear (asignar) tareas a procesos. Una estrategia de mapeado alternativa a la del Algoritmo 2.21 podría explotar esta información mapeando tareas que presentan un alto grado de interacción mutua al mismo proceso (procesador). Sin embargo, no se ha optado por esta alternativa en el caso del Algoritmo 2.21, al no tratarse de un factor de peso significativo en las prestaciones del algoritmo paralelo. La razón de esta afirmación se encuentra en la relación que existe entre el número de accesos "locales" y "remotos". Los accesos a memoria locales son aquellos que realiza un proceso a las estructuras de datos que se *crean* durante la ejecución de una tarea. Por ejemplo, los accesos a memoria que realiza el proceso encargado de ejecutar la tarea $(3, 2)$ para calcular (2.113) son locales, ya que las estructuras de datos empleadas para almacenar la matriz de entrada, $A^{(3,2)}$, y los factores incompletos, $\tilde{L}^{(3,2)}$ y $\tilde{D}^{(3,2)}$, son estructuras de datos que este proceso crea dinámicamente

durante la ejecución de la tarea $(3,2)$[5]. Por otra parte, los accesos a memoria remotos son los que realiza un proceso a estructuras de datos creadas, potencialmente, por otros procesos. Por ejemplo, los accesos a memoria que realiza el proceso encargado de ejecutar la tarea $(2,1)$, para recuperar $\hat{S}_C^{(3,1)}$ y $\hat{S}_C^{(3,2)}$ en la Figura 2.18, son remotos. En el algoritmo paralelo, *la parte más significativa de los accesos a memoria son locales, mientras que el número de accesos remotos es (relativamente) despreciable*. Esta propiedad del algoritmo paralelo proporciona libertad a la estrategia de planificación para equilibrar dinámicamente la carga computacional, ya que cualquier tarea puede ser potencialmente mapeada a cualquier proceso sin incurrir en un sobrecoste excesivo asociado al movimiento de datos entre procesos.

2.3.5. Ejemplo de aplicación a la EDP de difusión con coeficientes discontinuos

En este apartado analizamos el rendimiento del Algoritmo 2.21 y de la estrategia de planificación dinámica que éste incorpora, para calcular precondicionadores paralelos de sistemas de ecuaciones lineales que surgen en el contexto de la resolución numérica de la EDP de difusión en estado estacionario con coeficientes discontinuos. La EDP de difusión en estado estacionario con coeficientes discontinuos es una variante de (2.79) en la que los coeficientes de las derivadas parciales, $a(x,y)$ y $b(x,y)$, varían en función de la región del dominio computacional. En concreto, consideramos las cuatro alternativas que se ilustran en la Figura 2.20. Esta elección del valor de los coeficientes se ha efectuado *estratégicamente*, para crear diferentes escenarios de planificación para el Algoritmo 2.21. Los cuatro sistemas de ecuaciones de este apartado surgen, respectivamente, de la discretización, mediante diferencias finitas de segundo orden, de las cuatro EDPs mostradas en la Figura 2.20, usando una malla regular de 1000×1000 nodos; substituir $N = 1000$ en (2.82). Por lo tanto, para todos los casos considerados, la matriz de coeficientes del sistema es de dimensión $n = 10^6$, con 5 elementos no nulos por cada fila/columna.

La implementación del Algoritmo 2.20 corre a cargo de la biblioteca secuencial de particionado de grafos SCOTCH [128], versión 5.0.6RC16. La biblioteca se ha parametrizado de tal forma que el Algoritmo 2.20 emplea la ordenación natural para los separadores y los subgrafos independientes en las líneas 5 y 10, respectivamente; ver, por ejemplo, la numeración de los nodos en la Figura 2.10 (d). Además, se emplea el ajuste por defecto que define la biblioteca para los parámetros de la rutina calcular_separador[6]. En lo que respecta al Algoritmo 2.21, se ha desactivado el uso de heurísticos de reducción de llenado para la rutina calcular_factorización y, por las mismas razones especificadas en el Apartado 2.2.4, hemos escogido $\kappa = 3$ y $\tau = 10^{-3}$ para el núcleo de factorización con pivotamiento basado en la inversa que ésta (rutina) incorpora. Las computaciones numéricas se han efectuado con el formato de representación de números en coma flotante IEEE de doble precisión. Todos los resultados se han obtenido en el multiprocesador SGI Altix 350 (consultar el Apéndice B). El entorno de computación paralela se ajustó de tal forma que se ligó un proceso (hebra) por procesador físico del multiprocesador, y las hebras no migraron durante la ejecución del algoritmo paralelo.

Las Tablas 2.1, 2.2, 2.3 y 2.4 muestran, para los cuatro sistemas de ecuaciones considerados, la relación que existe entre el número de niveles/hojas del árbol tareas y el tiempo de ejecución paralelo, T_p (s), que se obtiene como resultado de aplicar la estrategia de planificación dinámica que incorpora el Algoritmo 2.21. El número de procesos involucrados en la computación paralela se especifica en la columna etiquetada como p. Los resultados mostrados para $p = 1$ se obtuvieron con el algoritmo que incorpora la biblioteca numérica ILUPACK, es decir, con el Algoritmo 2.15. La altura (número de niveles) y el número de hojas del árbol, se fijaron durante la fase de particionado como una función del valor del parámetro f; ver Algoritmo 2.20. El experimento

[5] En concreto, el proceso crea $A^{(3,2)}$ en la rutina construir_sumando, y $\tilde{L}^{(3,2)}$, $\tilde{D}^{(3,2)}$, en la rutina calcular_factorización.

[6] En la biblioteca SCOTCH, esta rutina implementa heurísticos multinivel [89, 102] para el cálculo de separadores de vértices. Para más detalles sobre los parámetros de este tipo de heurísticos y el valor por defecto que la biblioteca SCOTCH les asigna, consultar [130].

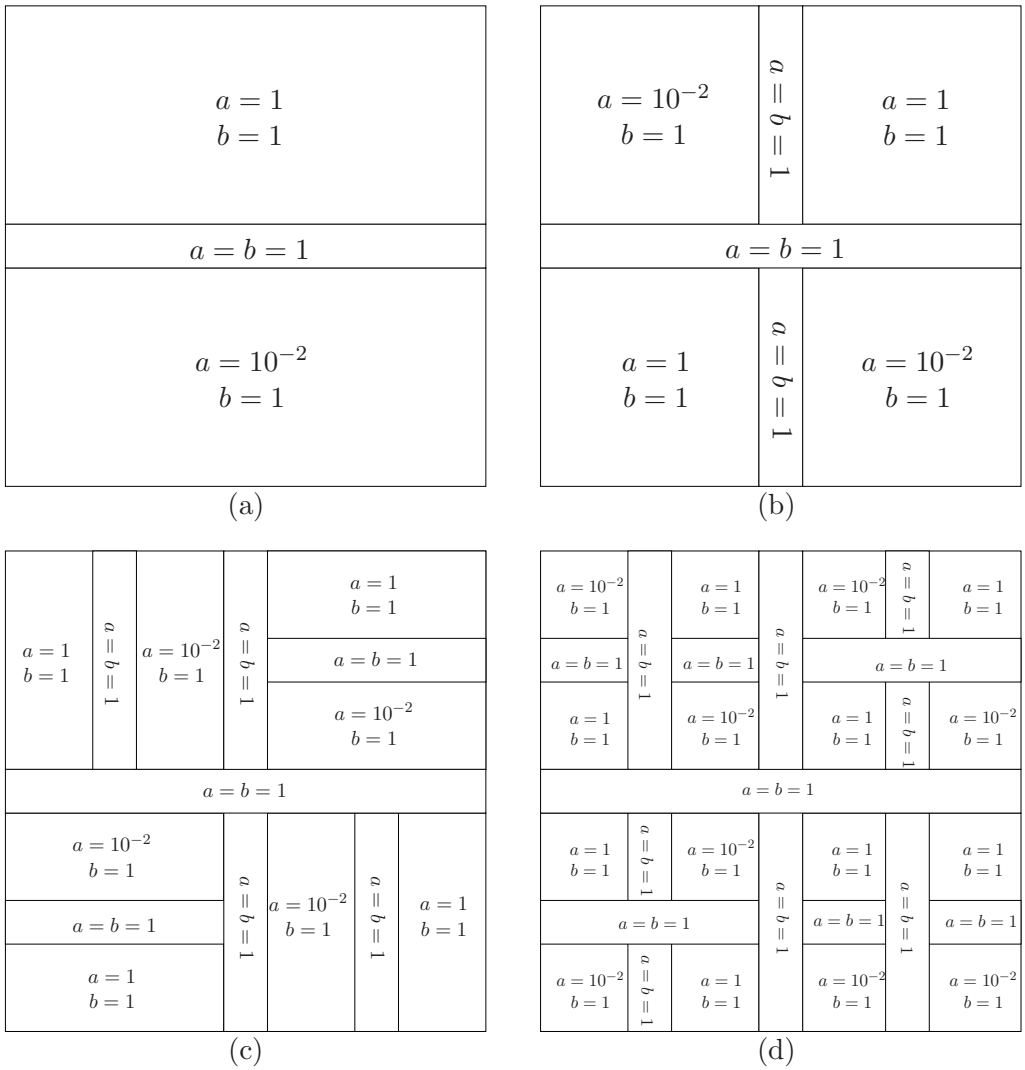


Figura 2.20: Cuatro variantes de la EDP de difusión en estado estacionario con coeficientes discontinuos. Las variantes se diferencian entre sí por la forma en la que están definidos sus coeficientes, $a(x, y)$, $b(x, y)$, en función de la región del dominio computacional: (a) 3 regiones; (b) 7 regiones; (c) 15 regiones; (c) 31 regiones.

explora diferentes valores del parámetro f en el intervalo $[p, 3p]$. En concreto, estos valores se obtienen multiplicando el número de procesos por el valor indicado en la columna etiquetada como f. Aquellos valores de f para los que se obtiene un árbol de tareas idéntico al valor de f inmediatamente inferior, como por ejemplo, $p = 2/f = p \times 1{,}50$, se han omitido de las tablas. Los resultados para $f = c$ se corresponden con una estrategia de particionado alternativa a la del Algoritmo 2.20, que obtiene árboles de tareas de altura $log_2(p) + 1$ *completos*, es decir, con p hojas. Como se puede observar en las Tablas 2.1, 2.2, 2.3 y 2.4, al incrementar el valor de f, se obtienen árboles de tareas de mayor altura y/o con mayor número de hojas. En general, esto permite reducir el tiempo de ejecución correspondiente al cálculo del precondicionador multinivel. Para revelar la causa principal de esta relación, la columna etiquetada como "$cv\,(\,\%)$" se refiere al *coeficiente de variación*, que se obtiene como el cociente entre la media aritmética y la desviación típica de la suma de los costes computacionales de las tareas que la estrategia de mapeado dinámica asigna a cada proceso. Un valor más pequeño de este coeficiente de variación indica una distribución más homogénea de la carga computacional. *Los resultados de las tablas enlazan claramente la reducción en el tiempo de ejecución con una distribución más equilibrada de la carga de trabajo.* La columna etiquetada como "Acel. $(\,\%)$" cuantifica esta reducción mostrando la aceleración relativa en el tiempo de ejecución que se obtiene empleando valores de f el intervalo $[p, 3p]$, con respecto al que se obtiene con $f = c$. De estos experimentos se pueden extraer dos conclusiones principales. En primer lugar, manipulando el valor de f en la fase de particionado se puede ajustar el grado de paralelismo disponible para la fase de construcción del precondicionador. En segundo lugar, el coste computacional del proceso de construcción del precondicionador paralelo está concentrado en las hojas, lo cual justifica las decisiones que se han tomado para la estrategia de planificación de tareas que incorpora el Algoritmo 2.21.

Los resultados marcados en **negrita** en las Tablas 2.1 y 2.3 corresponden a dos casos extremos. Para $p = 4/f = c$ en la Tabla 2.1, la carga computacional está desequilibrada ($cv = 43{,}9\,\%$), mientras que para $p = 4/f = c$ en la Tabla 2.3, la carga está bien repartida ($cv = 8{,}0\,\%$). Para analizar con más detalle la causa de estos resultados, la Figura 2.21 muestra la distribución y número de nodos aceptados y rechazados por el algoritmo paralelo en los cuatro primeros niveles del proceso de construcción del precondicionador. Estos cuatro primeros niveles pertenecen a la primera fase de la transformación del Algoritmo 2.15; ver Apartado 2.3.2. Los resultados de la Figura 2.21 se obtuvieron en un ordenador de sobremesa haciendo uso de MATLAB y, por tanto, se empleó una malla de tamaño moderado, $N = 200$. El comportamiento observado en la figura es representativo del caso de mayor dimensión, $N = 1000$. La columna izquierda de la Figura 2.21 se corresponde con el dominio computacional de la Figura 2.20 (a) y, la columna derecha, con el dominio computacional de la Figura 2.20 (c). Como se puede observar en la columna izquierda de la Figura 2.21, el separador raíz calculado por la fase de particionado coincide, aproximadamente, con la frontera de la región superior e inferior de la Figura 2.20 (a). Como consecuencia, dos tareas hoja del árbol se reparten la región superior y, las dos restantes, la región inferior. Dado que el método numérico tiende a aceptar *significativamente* mayor volumen de nodos en la región inferior que en la superior (ver columna izquierda de la Figura 2.21), la carga computacional está desequilibrada. Por otra parte, la columna derecha de la Figura 2.21 muestra que el separador raíz y los separadores del segundo nivel coinciden, aproximadamente, con las fronteras del dominio computacional de la Figura 2.18 (c). Como consecuencia, cada subgrafo independiente se corresponde, aproximadamente, con una región del dominio computacional compuesta por una primera submitad del tipo homogéneo ($a = 1$, $b = 1$), y una segunda submitad del tipo anisotrópico ($a = 10^{-2}$, $b = 1$). Dado que cada tarea hoja acepta y rechaza aproximadamente el mismo número de nodos al procesar el subgrafo independiente correspondiente (ver columna izquierda de la Figura 2.21), la carga está equilibrada. Cabe mencionar que ambas situaciones son fortuitas ya que, a la hora de calcular los separadores y los subgrafos independientes, la fase de particionado no es consciente de la distribución de los coeficientes de la EDP entre las regiones del dominio computacional. Nuestra estrategia gestiona esta incertidumbre obteniendo

		Árbol de tareas		Precondicionador paralelo LDL^T multinivel			
p	f	#niveles	#hojas	cv (%)	T_p (s)	Acel. (%)	nnz_M (mill.)
1	× 1,00	1	1	0,0	16,0	0,0	10,2
2	c	2	2	44,3	11,1	0,0	10,1
2	× 1,00	3	3	26,3	9,7	12,0	10,1
2	× 1,25	3	4	0,2	7,6	31,2	10,1
2	× 2,00	4	6	0,7	7,6	31,7	10,1
2	× 2,25	4	8	2,8	7,7	30,8	10,1
4	**c**	**3**	**4**	**43,9**	**5,6**	**0,0**	**10,1**
4	× 1,00	4	6	28,6	4,9	12,4	10,1
4	× 1,25	4	8	21,0	5,0	10,5	10,1
4	× 2,00	5	13	12,5	4,5	18,5	10,0
4	× 2,25	5	14	10,3	4,1	26,3	10,0
4	× 2,50	5	15	5,5	4,0	27,7	10,0
4	× 2,75	5	16	0,4	3,7	33,2	10,0
8	c	4	8	42,6	2,8	0,0	10,1
8	× 1,00	5	13	27,1	2,7	4,7	10,0
8	× 1,25	5	15	19,4	2,5	10,9	10,0
8	× 1,50	5	16	18,8	2,5	11,1	10,0
8	× 1,75	6	18	8,5	2,2	21,7	10,0
8	× 2,00	6	24	3,7	2,0	27,8	9,9
8	× 2,25	6	28	7,1	2,1	25,8	9,9
8	× 2,50	6	31	4,1	2,0	27,5	9,9
8	× 2,75	6	32	1,7	2,0	29,7	9,9
16	c	5	16	41,4	1,5	0,0	10,0
16	× 1,00	6	24	23,1	1,3	9,8	9,9
16	× 1,25	6	31	15,4	1,3	14,4	9,9
16	× 1,50	6	32	14,1	1,3	14,8	9,9
16	× 1,75	7	35	9,5	1,2	20,4	10,0
16	× 2,00	7	44	7,1	1,1	23,3	10,0
16	× 2,25	7	56	6,8	1,1	25,8	9,9
16	× 2,50	7	62	2,7	1,1	27,9	9,9
16	× 2,75	7	64	2,2	1,1	28,7	9,9
16	× 3,00	8	65	1,7	1,1	29,5	9,9

Tabla 2.1: Rendimiento de la estrategia de planificación dinámica cuando se aplica al sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (a).

		Árbol de tareas		Precondicionador paralelo LDL^T multinivel			
p	f	#niveles	#hojas	cv (%)	T_p (s)	Acel. (%)	nnz_M (mill.)
1	× 1,00	1	1	0,0	16,1	0,0	10,2
2	c	2	2	7,7	8,4	0,0	10,1
2	× 1,00	3	3	7,9	8,3	0,9	10,1
2	× 1,25	3	4	1,6	7,8	6,6	10,2
2	× 2,00	4	6	0,4	7,6	9,5	10,1
2	× 2,25	4	8	0,2	7,4	11,1	10,0
4	c	3	4	34,5	5,3	0,0	10,2
4	× 1,00	4	6	23,9	5,1	3,6	10,1
4	× 1,25	4	8	18,8	4,8	10,0	10,0
4	× 2,00	5	13	2,6	3,8	28,4	10,0
4	× 2,25	5	14	3,4	4,0	25,3	10,0
4	× 2,50	5	15	10,5	4,2	20,7	10,0
4	× 2,75	5	16	3,4	3,9	26,8	10,0
8	c	4	8	34,1	2,7	0,0	10,0
8	× 1,00	5	13	22,7	2,4	9,0	10,0
8	× 1,25	5	15	19,3	2,4	9,9	10,0
8	× 1,50	5	16	14,7	2,4	11,3	10,0
8	× 1,75	6	18	9,1	2,4	11,1	10,0
8	× 2,00	6	24	7,8	2,1	22,7	10,0
8	× 2,25	6	28	3,9	2,0	23,9	10,0
8	× 2,50	6	31	3,7	2,0	24,1	10,0
8	× 2,75	6	32	2,0	2,0	26,3	10,0
16	c	5	16	34,9	1,5	0,0	10,0
16	× 1,00	6	24	24,4	1,4	4,5	10,0
16	× 1,25	6	31	15,3	1,2	17,1	10,0
16	× 1,50	6	32	13,4	1,2	16,8	10,0
16	× 1,75	7	35	9,8	1,2	21,3	10,0
16	× 2,00	7	44	7,3	1,1	22,1	10,0
16	× 2,25	7	56	6,3	1,1	23,7	9,9
16	× 2,50	7	62	4,8	1,1	22,6	9,9
16	× 2,75	7	64	3,2	1,1	23,4	9,9
16	× 3,00	8	65	3,4	1,1	23,0	9,9

Tabla 2.2: Rendimiento de la estrategia de planificación dinámica cuando se aplica al sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (b).

		Árbol de tareas		Precondicionador paralelo LDL^T multinivel			
p	f	#niveles	#hojas	cv (%)	T_p (s)	Acel. (%)	nnz_M (mill.)
1	× 1,00	1	1	0,0	16,1	0,0	10,1
2	c	2	2	7,5	8,3	0,0	10,1
2	× 1,00	3	3	7,2	8,3	0,7	10,1
2	× 1,25	3	4	1,7	7,8	6,1	10,2
2	× 2,00	4	6	2,1	7,7	7,1	10,1
2	× 2,25	4	8	6,8	8,0	4,0	10,1
4	**c**	**3**	**4**	**8,0**	**4,3**	**0,0**	**10,2**
4	× 1,00	4	6	5,1	4,1	4,4	10,1
4	× 1,25	4	8	10,5	4,2	3,1	10,1
4	× 2,00	5	13	7,6	4,2	4,0	10,1
4	× 2,25	5	14	6,6	4,1	5,3	10,1
4	× 2,50	5	15	5,0	4,0	7,4	10,0
4	× 2,75	5	16	2,7	3,8	11,8	10,0
8	c	4	8	26,3	2,8	0,0	10,1
8	× 1,00	5	13	19,2	2,4	12,4	10,1
8	× 1,25	5	15	16,9	2,5	11,1	10,0
8	× 1,50	5	16	14,1	2,4	12,4	10,0
8	× 1,75	6	18	9,4	2,2	21,5	10,0
8	× 2,00	6	24	9,2	2,2	22,5	10,0
8	× 2,25	6	28	4,7	2,1	26,1	10,0
8	× 2,50	6	31	4,1	2,1	26,1	10,0
8	× 2,75	6	32	3,7	2,0	27,4	10,0
16	c	5	16	30,3	1,4	0,0	10,0
16	× 1,00	6	24	20,7	1,3	7,8	10,0
16	× 1,25	6	31	13,2	1,2	12,7	10,0
16	× 1,50	6	32	10,0	1,2	13,5	10,0
16	× 1,75	7	35	8,3	1,2	16,8	10,0
16	× 2,00	7	44	5,8	1,1	20,5	10,0
16	× 2,25	7	56	7,6	1,2	18,3	9,9
16	× 2,50	7	62	4,4	1,1	22,9	9,9
16	× 2,75	7	64	2,6	1,1	23,1	9,9
16	× 3,00	8	65	2,9	1,1	21,5	9,9

Tabla 2.3: Rendimiento de la estrategia de planificación dinámica cuando se aplica al sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (c).

		Árbol de tareas		Precondicionador paralelo LDL^T multinivel			
p	f	#niveles	#hojas	cv (%)	T_p (s)	Acel. (%)	nnz_M (mill.)
1	× 1,00	1	1	0,0	16,2	0,0	10,2
2	c	2	2	7,4	8,3	0,0	10,0
2	× 1,00	3	3	7,6	8,2	1,2	10,1
2	× 1,25	3	4	2,0	7,9	5,4	10,2
2	× 2,00	4	6	0,8	7,7	7,7	10,1
2	× 2,25	4	8	1,7	7,6	8,4	10,1
4	c	3	4	9,9	4,3	0,0	10,2
4	× 1,00	4	6	8,3	4,1	5,2	10,1
4	× 1,25	4	8	5,0	4,1	5,8	10,1
4	× 2,00	5	13	2,5	3,9	10,6	10,1
4	× 2,25	5	14	2,1	3,9	10,4	10,1
4	× 2,50	5	15	3,5	3,8	12,0	10,1
4	× 2,75	5	16	3,7	3,9	10,6	10,1
8	c	4	8	15,7	2,4	0,0	10,1
8	× 1,00	5	13	13,5	2,2	7,6	10,1
8	× 1,25	5	15	11,2	2,1	13,6	10,1
8	× 1,50	5	16	9,6	2,2	8,6	10,1
8	× 1,75	6	18	8,8	2,3	6,4	10,1
8	× 2,00	6	24	4,2	2,0	16,7	10,1
8	× 2,25	6	28	3,2	2,0	17,7	10,0
8	× 2,50	6	31	5,3	2,1	12,9	10,0
8	× 2,75	6	32	5,1	2,1	14,8	10,0
16	c	5	16	24,5	1,4	0,0	10,1
16	× 1,00	6	24	18,0	1,4	1,4	10,1
16	× 1,25	6	31	10,8	1,2	12,0	10,0
16	× 1,50	6	32	10,7	1,2	12,7	10,0
16	× 1,75	7	35	9,3	1,2	14,8	10,0
16	× 2,00	7	44	9,9	1,1	19,4	10,0
16	× 2,25	7	56	5,0	1,1	21,2	10,0
16	× 2,50	7	62	3,8	1,1	22,0	10,0
16	× 2,75	7	64	3,8	1,1	21,2	10,0
16	× 3,00	8	65	2,7	1,1	21,2	10,0

Tabla 2.4: Rendimiento de la estrategia de planificación dinámica cuando se aplica al sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (d).

un mayor número de subgrafos independientes que de procesos y mejorando, en tiempo de ejecución, el equilibrio de la carga computacional mediante la expresión de planificación dinámica que incorpora el Algoritmo 2.21; ver Tablas 2.1 y 2.4.

La columna etiquetada como "nnz_M (mill.)" en las Tablas 2.1, 2.2, 2.3 y 2.4 se refiere al número de millones de elementos no nulos del precondicionador multinivel. Este indicador sirve como estimador del número de operaciones en coma flotante que serán necesarias para aplicar el precondicionador en la fase de resolución del sistema precondicionado. En el caso del Algoritmo 2.15, el número de elementos no nulos del precondicionador multinivel se obtiene de forma acorde a la siguiente expresión

$$nnz_M = \sum_{i}^{l-1} \left[nnz(tril(\tilde{L}_B^i)) + nnz(\tilde{D}_B^i) + nnz(F^i) \right] + nnz(tril(\tilde{L}_B^l)) + nnz(\tilde{D}_B^l), \qquad (2.124)$$

donde l es el número de niveles que construye el algoritmo numérico, $\tilde{L}_B$, $\tilde{D}_B$ y F se refieren a los bloques de la factorización aproximada (2.77), y el superíndice de dichos bloques identifica el nivel correspondiente del proceso de construcción del precondicionador. La función $tril$ devuelve el triángulo inferior estricto de la matriz que recibe como entrada, y la función nnz, su número de elementos no nulos. La expresión correspondiente al Algoritmo 2.21 es una generalización de (2.124), que tiene en cuenta la factorización multinivel que calcula localmente cada tarea del árbol. Como se puede observar en las Tablas 2.1, 2.2, 2.3 y 2.4, el número de elementos no nulos del precondicionador construido por el Algoritmo 2.15 (resultados para $p = 1$ en las tablas) es cercano al número de elementos no nulos de los precondicionadores construidos por el Algoritmo 2.21 (resultados para $p = 2, 4, 8$ y 16 en las tablas). En el capítulo de resultados computacionales proporcionamos evidencias adicionales que refuerzan nuestra proposición de que nuestro enfoque para la paralelización del proceso de construcción del precondicionador preserva la semántica del algoritmo numérico de la biblioteca ILUPACK.

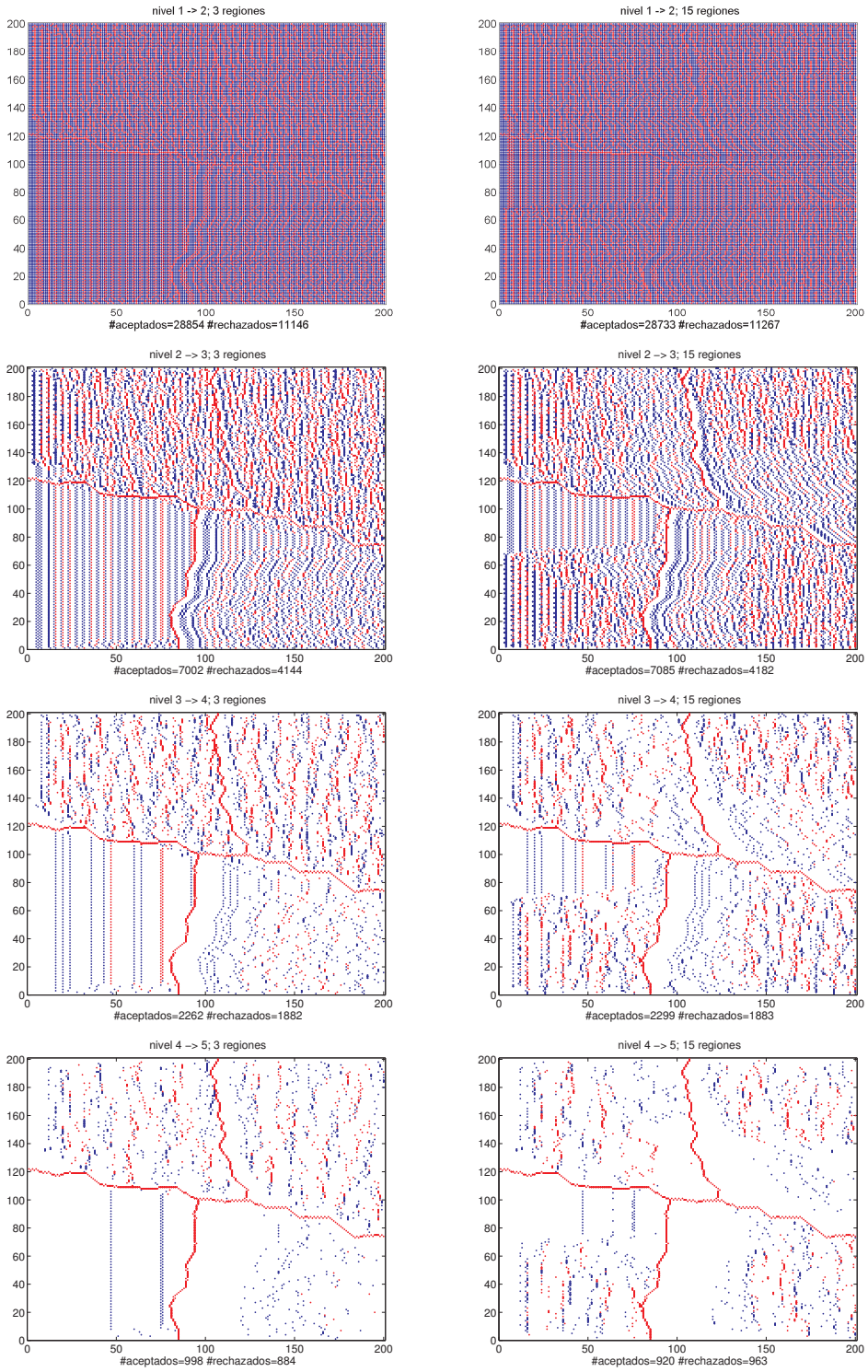


Figura 2.21: Número y distribución de las incógnitas aceptadas (puntos azules) y rechazadas (puntos rojos) en los primeros cuatro niveles de la primera fase de la transformación del Algoritmo 2.15. Izquierda: dominio computacional con 3 regiones; ver Figura 2.18 (a). Derecha: dominio computacional con 15 regiones; ver Figura 2.18 (c).

Capítulo 3

Resolución iterativa paralela del sistema

En este capítulo presentamos nuestro enfoque para la paralelización del método de los gradientes conjugados precondicionados (*Preconditioned Conjugate Gradients*, PCG) mediante los precondicionadores paralelos basados en la factorización LDL^T multinivel. Este enfoque aprovecha, en la *aplicación* de estos precondicionadores, el paralelismo de tareas habilitado de forma natural por su estructura lógica. La paralelización del resto de operaciones del método PCG, es decir, productos escalares, actualizaciones de vectores y producto de una matriz dispersa por un vector, se realiza de manera coordinada con la aplicación del precondicionador, con el objeto de aumentar la localidad de referencia y reducir el sobrecoste asociado al tráfico de datos en el sistema de memoria del multiprocesador.

La estructura del capítulo es la siguiente. En la Sección 3.1 revisamos el método de los gradientes conjugados, sus propiedades, fundamentos, estructura, convergencia y precondicionado. A continuación, en la Sección 3.2 discutimos la aplicación de precondicionadores basados en la factorización LDL^T multinivel, y detalles de su implementación en la biblioteca ILUPACK. Posteriormente, en la Sección 3.3 se analiza la concurrencia que se puede explotar en la aplicación de la versión paralela de estos precondicionadores, se indica cómo aprovechar esta concurrencia de forma eficiente, y se presentan las estrategias de mapeado y planificación de las tareas en las que se descompone la aplicación del precondicionador. La paralelización del resto de operaciones del método PCG es objeto de la Sección 3.4, que también estudia el rendimiento del método paralelo de los gradientes conjugados precondicionados cuando se aplica para resolver los sistemas de ecuaciones que surgen en la aplicación de ejemplo empleada en el Capítulo 2.

3.1. El método de los gradientes conjugados

El método de los gradientes conjugados (*Conjugate Gradients*, CG) es un método iterativo para la resolución de sistemas de ecuaciones simétricos definidos positivos (SDP) desarrollado por Hestenes y Stiefel en [93]. En dicho trabajo, el método se derivó explotando principios del campo de la minimización de formas cuadráticas. Simultáneamente, Lanczos [108] desarrolló un algoritmo que, cuando se aplica para la resolución de sistemas de ecuaciones lineales SDP, es *matemáticamente equivalente* al algoritmo CG. En esta tesis nos centramos en la formulación del método CG desarrollada por Hestenes y Stiefel en [93] ya que, entre otras razones, es la que incorpora la biblioteca ILUPACK para la resolución de sistemas de ecuaciones lineales SDP.

Esta sección está estructurada de la forma siguiente. En primer lugar, el Apartado 3.1.1 deriva paulatinamente el método y presenta sus fundamentos teóricos. Posteriormente, el Apartado 3.1.2 revisa los factores que condicionan la convergencia del método, motivando el uso de las técnicas de precondicionado. Por último, el Apartado 3.1.3 presenta cómo transformar el método básico para resolver el sistema precondicionado de forma eficiente.

3.1.1. Derivación y fundamentos

Formas cuadráticas, direcciones de búsqueda y búsquedas lineales exactas

Una *forma cuadrática* es una función $\phi : \mathbb{R}^n \to \mathbb{R}$ de la forma

$$\phi(y) = \frac{1}{2} y^T A y - y^T b, \tag{3.1}$$

donde $A \in \mathbb{R}^{n,n}$, e $y, b \in \mathbb{R}^n$. Si A es una matriz SDP, entonces la solución del sistema $Ax = b$ minimiza la forma cuadrática ϕ, es decir,

$$\phi(x) = \min_{y \in \mathbb{R}^n} \phi(y). \tag{3.2}$$

Por lo tanto, resolver $Ax = b$ y minimizar $\phi(y)$ son problemas equivalentes. Para demostrar esta conexión, partimos de la expresión correspondiente al gradiente de $\phi(y)$

$$\phi'(y) = \begin{pmatrix} \frac{\partial}{y_1}\phi(y) \\ \vdots \\ \frac{\partial}{y_n}\phi(y) \end{pmatrix} = Ay - b, \tag{3.3}$$

donde $Ay - b$ es el vector residuo (cambiado de signo) asociado a y en el sistema $Ax = b$. El mínimo de $\phi(y)$ es un vector que anula el vector gradiente $\phi'(y)$. Por lo tanto, por (3.3), el único vector que anula el vector gradiente, es decir, el único vector que minimiza $\phi(y)$, es la solución de $Ax = b$.

El método de los gradientes conjugados parte de una solución inicial $x^{(0)}$, y construye una secuencia de aproximaciones $x^{(1)}, x^{(2)}, \ldots$ al mínimo de $\phi(y)$. Para cada una de estas aproximaciones, el *error* $e^{(i)} = x^{(i)} - x$ y el *residuo* $r^{(i)} = b - Ax^{(i)}$ son dos vectores que indican lo lejos que está el proceso de aproximación de x y b, respectivamente. Ambos vectores están relacionados por la igualdad $r^{(i)} = -Ae^{(i)}$.

Con el propósito de minimizar $\phi(y)$ progresivamente, cada iteración del método construye $x^{(i+1)}$ a partir de $x^{(i)}$, de tal forma que $\phi(x^{(i+1)}) < \phi(x^{(i)})$, para $i = 0, 1, \ldots$. Este proceso combina dos ingredientes básicos: (1) selección de una dirección de búsqueda; y (2) búsqueda lineal. La selección de una dirección de búsqueda consiste en encontrar un vector, $p^{(i)}$, que marca la dirección a través de la cual $x^{(i+1)}$ se obtiene a partir de $x^{(i)}$. Una vez que se ha encontrado $p^{(i)}$, entonces $x^{(i+1)}$ se selecciona dentro del conjunto $\{x^{(i)} + \alpha p^{(i)} : \alpha \in \mathbb{R}\}$. Por lo tanto, se cumple que

$$x^{(i+1)} = x^{(i)} + \alpha^{(i)} p^{(i)}, \text{ con } i = 0, 1, \ldots, \tag{3.4}$$

para algún escalar $\alpha^{(i)}$ y, además, $r^{(i+1)}$ se puede construir a partir de $r^{(i)}$ conforme a la recurrencia

$$r^{(i+1)} = -Ae^{(i+1)} = -A(e^{(i)} + \alpha^{(i)} p^{(i)}) = r^{(i)} - \alpha^{(i)} A p^{(i)}. \tag{3.5}$$

El proceso que escoge $\alpha^{(i)}$ entre todos los escalares $\alpha \in \mathbb{R}$, se denomina *búsqueda lineal*. Entre todas las posibles opciones, el método CG escoge $\alpha^{(i)}$ de modo que

$$\phi(x^{(i)} + \alpha^{(i)} p^{(i)}) = \min_{\alpha \in \mathbb{R}} \phi(x^{(i)} + \alpha p^{(i)}), \tag{3.6}$$

ya que de esta forma $\phi(x^{(i+1)}) \leq \phi(x^{(i)})$. Este tipo de búsqueda lineal se denomina *búsqueda lineal exacta*. Para obtener la solución a (3.6), calculamos la derivada direccional

$$\frac{d}{d\alpha}\phi(x^{(i+1)}) = \phi'(x^{(i+1)})^T \frac{d}{d\alpha} x^{(i+1)} = \phi'(x^{(i+1)})^T p^{(i)}, \tag{3.7}$$

donde $\phi'(x^{(i+1)}) = -r^{(i+1)}$ (ver (3.3)). Seguidamente, igualando a cero

$$-(r^{(i+1)})^T p^{(i)} = (p^{(i)})^T A e^{(i+1)} = (p^{(i)})^T A(e^{(i)} + \alpha^{(i)} p^{(i)}) = 0, \tag{3.8}$$

obtenemos que

$$\alpha^{(i)} = -\frac{(p^{(i)})^T A e^{(i)}}{(p^{(i)})^T A p^{(i)}} = \frac{(p^{(i)})^T r^{(i)}}{(p^{(i)})^T A p^{(i)}}. \tag{3.9}$$

Con esta elección del valor de $\alpha^{(i)}$, se cumple que

$$\phi(x^{(i+1)}) = \phi(x^{(i)}) - \frac{1}{2}\frac{((p^{(i)})^T r^{(i)})^2}{(p^{(i)})^T A p^{(i)}}, \tag{3.10}$$

y, por tanto, para garantizar una reducción en el valor de $\phi(y)$, el producto escalar $(p^{(i)})^T r^{(i)}$ no puede ser nulo, es decir, $p^{(i)}$ *no puede ser ortogonal a* $r^{(i)}$.

Direcciones de búsqueda A-ortogonales

En el método CG, las direcciones de búsqueda son A-ortogonales o conjugadas. Una secuencia de direcciones de búsqueda $p^{(0)}, p^{(1)}, \ldots, p^{(i)}$ cumple esta propiedad si y solo si

$$(p^{(k)})^T A p^{(j)} = 0, \ \textbf{con}\ k = 1, 2, \ldots i, \quad j = 1, 2, \ldots k-1, \quad k \neq j. \tag{3.11}$$

Esta decisión tiene consecuencias *estratégicas* sobre las propiedades de la secuencia de aproximaciones calculada por el algoritmo:

1. *El proceso resultante de aplicar la recurrencia (3.4), con* $i = 0, 1, 2, \ldots, n-1$, *converge a la solución del sistema* $Ax = b$. Para demostrar esta propiedad, comenzamos considerando el error inicial $e^{(0)}$ como una combinación lineal de las direcciones de búsqueda

$$e^{(0)} = \sum_{j=0}^{n-1} \delta^{(j)} p^{(j)}. \tag{3.12}$$

Multiplicando ambas partes de esta igualdad por $(p^{(i)})^T A$, con $i = 0, 1, \ldots, n-1$, entonces

$$(p^{(i)})^T A e^{(0)} = \sum_{j=0}^{n-1} \delta^{(j)} (p^{(i)})^T A p^{(j)} = \delta^{(i)} (p^{(i)})^T A p^{(i)}, \tag{3.13}$$

y, despejando $\delta^{(i)}$, finalmente obtenemos

$$\delta^{(i)} = \frac{(p^{(i)})^T A e^{(0)}}{(p^{(i)})^T A p^{(i)}} = \frac{(p^{(i)})^T A (e^{(0)} + \sum_{j=0}^{i-1} \alpha^{(j)} p^{(j)})}{(p^{(i)})^T A p^{(i)}} = \frac{(p^{(i)})^T A e^{(i)}}{(p^{(i)})^T A p^{(i)}}. \tag{3.14}$$

Por lo tanto, $\delta^{(i)} = -\alpha^{(i)}$, para $i = 0, 1, \ldots, n-1$ (relacionar (3.14) con (3.9)). Explotando esta relación entre $\delta^{(i)}$ y $\alpha^{(i)}$ es posible comprobar que $e^{(i)}$ se obtiene como resultado de eliminar las primeras i componentes del error inicial $e^{(0)}$

$$e^{(i)} = e^{(0)} + \sum_{j=0}^{i-1} \alpha^{(j)} p^{(j)} = \sum_{j=0}^{n-1} \delta^{(j)} p^{(j)} - \sum_{j=0}^{i-1} \delta^{(j)} p^{(j)} = \sum_{j=i}^{n-1} \delta^{(j)} p^{(j)}. \tag{3.15}$$

Tras n iteraciones, $e^{(n)} = 0$ o, equivalentemente, $x^{(n)}$ es la solución de $Ax = b$.

2. *La aproximación* $x^{(i)}$ *es la "más cercana" a la solución del sistema entre todos los vectores del subespacio* $\mathcal{D}_i = x^{(0)} + \text{span}\{p^{(0)}, p^{(1)}, \ldots, p^{(i-1)}\}$. En particular, la A-norma del error correspondiente a la aproximación construida en la iteración i-ésima, es decir, $\|e^{(i)}\|_A = \|x^{(i)} - x\|_A$ cumple

$$\|e^{(i)}\|_A = \min_{y \in \mathcal{D}_i} \|e_y\|_A, \tag{3.16}$$

con $e_y = y - x$ el error asociado a un vector arbitrario $y \in \mathcal{D}_i$. Para demostrar (3.16), consideramos la expansión de un vector y arbitrario de $\mathcal{D}_i$

$$y = x^{(0)} + \sum_{j=0}^{i-1} \sigma^{(j)} p^{(j)}, \tag{3.17}$$

y la A-norma del error asociado

$$\begin{aligned} \|e_y\|_A &= \|x^{(0)} + \sum_{j=0}^{i-1} \sigma^{(j)} p^{(j)} - x\|_A = \|e^{(0)} + \sum_{j=0}^{i-1} \sigma^{(j)} p^{(j)}\|_A \\ &= \|\sum_{j=0}^{n-1} \delta^{(j)} p^{(j)} + \sum_{j=0}^{i-1} \sigma^{(j)} p^{(j)}\|_A = \|\sum_{j=0}^{i-1} (\delta^{(j)} + \sigma^{(j)}) p^{(j)} + \sum_{j=i}^{n-1} \delta^{(j)} p^{(j)}\|_A \\ &= \|e^{(i)} + \sum_{j=0}^{i-1} (\delta^{(j)} + \sigma^{(j)}) p^{(j)}\|_A. \end{aligned}$$

Aplicando la definición de la A-norma a esta última expresión obtenemos

$$\begin{aligned} \|e_y\|_A &= \left((e^{(i)})^T + \sum_{j=0}^{i-1} (\delta^{(j)} + \sigma^{(j)}) (p^{(j)})^T \right) A \left(e^{(i)} + \sum_{j=0}^{i-1} (\delta^{(j)} + \sigma^{(j)}) p^{(j)} \right) \\ &= (e^{(i)})^T A e^{(i)} + \sum_{j=0}^{i-1} \sum_{k=0}^{i-1} \left[(\sigma^{(j)} + \delta^{(j)}) (\sigma^{(k)} + \delta^{(k)}) (p^{(j)})^T A p^{(k)} \right] \\ &= \|e^{(i)}\|_A + \sum_{j=0}^{i-1} (\sigma^{(j)} + \delta^{(j)})^2 (p^{(j)})^T A p^{(j)}. \end{aligned}$$

Por lo tanto, dado que $\|e_y\|_A$ contiene $\|e^{(i)}\|_A$, y que $(\sigma^{(j)} + \delta^{(j)})^2 (p^{(j)})^T A p^{(j)} \geq 0$, para $j = 0, 1, \ldots, i-1$, entonces $x^{(i)}$ cumple (3.16).

3. *El vector de error $e^{(i)}$ y el residuo $r^{(i)}$ son, respectivamente, A-ortogonal y ortogonal a las direcciones de búsqueda $p^{(k)}$, para $k = 0, 1, \ldots, i-1$.* Estas propiedades se pueden demostrar multiplicando ambas partes de la igualdad (3.15) por $Ap^{(k)}$

$$(e^{(i)})^T A p^{(k)} = -(r^{(i)})^T p^{(k)} = \sum_{j=i}^{n-1} \delta^{(j)} (p^{(j)})^T A p^{(k)} = 0, \ \textbf{con } k = 0, 1, \ldots, i-1. \tag{3.18}$$

A-conjugación de Gram-Schmidt

El método CG calcula direcciones de búsqueda A-ortogonales mediante un proceso conocido como *A-conjugación de Gram-Schmidt*. Tomando como entrada un conjunto de vectores $u_0, u_1, \ldots, u_{n-1}$ linealmente independientes, y partiendo de $p^{(0)} = u_0$, la iteración i-ésima de este proceso construye la dirección de búsqueda $p^{(i)}$ como

$$p^{(i)} = u_i + \sum_{j=0}^{i-1} \beta_{ij} p^{(j)}, \tag{3.19}$$

donde los coeficientes β_{ij} se obtienen para que $p^{(i)}$ sea A-ortogonal a las direcciones de búsqueda construidas en iteraciones previas, es decir, a las direcciones $p^{(j)}$, con $0 \leq j < i$. Teniendo en

cuenta que este conjunto de direcciones es A-ortogonal, y multiplicando ambas partes de la igualdad (3.19) por $Ap^{(k)}$, con $0 \leq k < i$, entonces

$$(p^{(i)})^T Ap^{(k)} = u_i^T Ap^{(k)} + \sum_{j=0}^{i-1} \beta_{ij} (p^{(j)})^T Ap^{(k)} = u_i^T Ap^{(k)} + \beta_{ik} (p^{(k)})^T Ap^{(k)}. \tag{3.20}$$

Finalmente, los coeficientes β_{ik} se obtienen fijando $(p^{(i)})^T Ap^{(k)} = 0$,

$$\beta_{ik} = -\frac{u_i^T Ap^{(k)}}{(p^{(k)})^T Ap^{(k)}}. \tag{3.21}$$

A no ser que los vectores $u_0, u_1, \ldots, u_{n-1}$ se escojan *estratégicamente*, el proceso de A-conjugación efectúa $O(n^3)$ operaciones aritméticas y, además, para construir una nueva dirección de búsqueda, necesita almacenar todas las direcciones de búsqueda previas. La clave para evitar estos inconvenientes es escoger $u_i = r^{(i)}$, para $i = 0, 1, \ldots, n-1$. Con esta elección, se cumple que

$$\text{span}\{r^{(0)}, r^{(1)}, \ldots, r^{(i-1)}\} = \text{span}\{p^{(0)}, p^{(1)}, \ldots, p^{(i-1)}\} = \mathcal{K}_i(A, r_0), \tag{3.22}$$

donde $\mathcal{K}_k(A, r^{(0)}) = \text{span}\{r^{(0)}, Ar^{(0)}, A^2 r^{(0)}, \ldots, A^{i-1} r^{(0)}\}$ es el subespacio de Krylov de dimensión i generado por la matriz A y el residuo inicial $r^{(0)}$ (consultar [76, 164] para la demostración). Además, $(r^{(i)})^T$ es A-ortogonal a $p^{(k)}$, para $k = 0, 1, \ldots, i-2$ y, por tanto,

$$\beta_{ik} = -\frac{(r^{(i)})^T Ap^{(k)}}{(p^{(k)})^T Ap^{(k)}} = 0, \tag{3.23}$$

para $k = 0, 1, \ldots, i-2$. Esta última propiedad se deriva partiendo de (3.22). Cualquier vector $p^{(k)} \in \text{span}\{p^{(0)}, \ldots, p^{(k)}\} = \text{span}\{r^{(0)}, Ar^{(0)}, \ldots, A^k r^{(0)}\}$ y, por tanto, $Ap^{(k)} \subseteq \mathcal{K}_{k+2}(A, r^{(0)}) = \text{span}\{r^{(0)}, \ldots, r^{(k+1)}\}$. Dado que $r^{(i)}$ es ortogonal a $r^{(0)}, r^{(1)}, \ldots, r^{(k+1)}$, para $k = 0, \ldots, i-2$, entonces $r^{(i)}$ es A-ortogonal a $p^{(k)}$, para $k = 0, 1, \ldots, i-2$.

Substituyendo (3.23) en (3.19), obtenemos

$$p^{(i)} = r^{(i)} + \sum_{j=0}^{i-1} \beta_{ij} p^{(j)} = r^{(i)} + \beta_{i,i-1}\, p^{(i-1)} = r^{(i)} - \frac{(r^{(i)})^T Ap^{(i-1)}}{(p^{(i-1)})^T Ap^{(i-1)}}\, p^{(i-1)}. \tag{3.24}$$

Por lo tanto, ya no es necesario almacenar las direcciones de búsqueda previas para garantizar la A-ortogonalidad de las nuevas direcciones. En esta crucial propiedad radica la elevada eficiencia computacional y las reducidas necesidades de almacenamiento del método de los gradientes conjugados.

El método

Para finalizar con la derivación del método CG, simplificamos la expresión para $\beta^{(i)} = \beta_{i,i-1}$. Para ello, en primer lugar, multiplicamos (3.24) por $r^{(i)}$, obteniendo la igualdad

$$(p^{(i)})^T r^{(i)} = (r^{(i)})^T r^{(i)} + \beta^{(i)} (p^{(i-1)})^T r^{(i)} = (r^{(i)})^T r^{(i)}. \tag{3.25}$$

En segundo lugar, calculando el producto escalar de $r^{(i)}$ y (3.5), obtenemos

$$(r^{(i)})^T r^{(j+1)} \quad = \quad (r^{(i)})^T r^{(j)} - \alpha^{(j)} (r^{(i)})^T Ap^{(j)}, \tag{3.26}$$

y, despejando $(r^{(i)})^T Ap^{(j)}$, entonces

$$(r^{(i)})^T Ap^{(j)} \quad = \quad \frac{1}{\alpha^{(j)}} ((r^{(i)})^T r^{(j)} - (r^{(i)})^T r^{(j+1)}).$$

Evaluando esta última expresión con $j = i - 1$, entonces

$$\begin{aligned}(r^{(i)})^T Ap^{(i-1)} &= \frac{1}{\alpha^{(i-1)}}((r^{(i)})^T r^{(i-1)} - (r^{(i)})^T r^{(i)}) \\ &= -\frac{1}{\alpha^{(i-1)}}(r^{(i)})^T r^{(i)} \\ &= -\frac{(p^{(i-1)})^T Ap^{(i-1)}}{(p^{(i-1)})^T r^{(i-1)}}(r^{(i)})^T r^{(i)}.\end{aligned}$$

Por último, teniendo en cuenta esta igualdad y (3.25), encontramos finalmente la expresión para $\beta^{(i)} = \beta_{i,i-1}$ en el método CG

$$\beta^{(i)} = -\frac{(r^{(i)})^T Ap^{(i-1)}}{(p^{(i-1)})^T Ap^{(i-1)}} = \frac{(r^{(i)})^T r^{(i)}}{(p^{(i-1)})^T r^{(i-1)}} = \frac{(r^{(i)})^T r^{(i)}}{(r^{(i-1)})^T r^{(i-1)}}. \tag{3.27}$$

Cabe destacar que, a partir de la igualdad (3.25), si $r^{(i)} \neq 0$, entonces $p^{(i)}$ no es ortogonal a $r^{(i)}$. Teniendo en cuenta (3.10), entonces cada iteración del método CG reduce *estrictamente* el valor de la función objetivo mientras no se alcance la solución del sistema, es decir, $\phi(x^{(i+1)}) < \phi(x^{(i)})$ mientras $x^{(i+1)}$ no sea la solución de $Ax = b$.

El método CG se muestra en el Algoritmo 3.1. La línea 5 de este algoritmo se obtiene combinando las expresiones (3.9) y (3.25), y las líneas 6, 7, 8, y 9, directamente a partir de (3.4), (3.5), (3.27) y (3.24), respectivamente.

Algoritmo 3.1: Método de los gradientes conjugados-CG.

1 Fijar una solución inicial $x^{(0)}$
2 $r^{(0)} \leftarrow b - Ax^{(0)}$
3 $p^{(0)} \leftarrow r^{(0)}$
4 **para** $i = 0, 1, \ldots,$ *hasta convergencia* **hacer**
5 $\quad \alpha^{(i)} \leftarrow \frac{(r^{(i)})^T r^{(i)}}{(p^{(i)})^T Ap^{(i)}}$
6 $\quad x^{(i+1)} \leftarrow x^{(i)} + \alpha^{(i)} p^{(i)}$
7 $\quad r^{(i+1)} \leftarrow r^{(i)} - \alpha^{(i)} Ap^{(i)}$
8 $\quad \beta^{(i+1)} \leftarrow \frac{(r^{(i+1)})^T r^{(i+1)}}{(r^{(i)})^T r^{(i)}}$
9 $\quad p^{(i+1)} \leftarrow r^{(i+1)} + \beta^{(i+1)} p^{(i)}$
10 **fin para**

La implementación eficiente del método CG se recoge en el Algoritmo 3.2. En este algoritmo, los superíndices de los vectores y escalares se han omitido para simplificar la notación y, además, se han añadido un nuevo vector q que almacena el producto de A por p y tres nuevos escalares, σ_1, σ_2 y δ que contienen, en la iteración i-ésima, los productos escalares $(r^{(i)})^T r^{(i)}$, $(r^{(i+1)})^T r^{(i+1)}$ y $(p^{(i)})^T Ap^{(i)}$, respectivamente. El Algoritmo 3.2 requiere almacenamiento para la matriz A y cuatro vectores (x, p, q y r), y efectúa tres tipos básicos de operaciones: (a) actualizaciones de vectores; (b) productos escalares; y (c) multiplicación de una matriz dispersa por un vector. Las operaciones del tipo (a) y (b) son de la forma $y \leftarrow y + \alpha x$ y $\beta \leftarrow y^T x$, respectivamente, donde x e y son vectores, y α y β, escalares. Las operaciones del tipo (c) son de la forma $y \leftarrow Ax$, donde A es una matriz dispersa. En concreto, sólo requiere tres operaciones del tipo (a), dos del tipo (b) y una del tipo (c).

3.1.2. Convergencia

El método CG fue originalmente clasificado como un método directo ya que, en aritmética exacta, calcula la solución del sistema en n iteraciones. En aritmética finita, sin embargo,

Algoritmo 3.2: Implementación eficiente del método de los gradientes conjugados-CG.

```
1  Fijar una solución inicial x
2  q ← Ax
3  r ← b − q
4  σ_1 ← r^T r
5  p ← r
6  para i = 0, 1, ..., hasta convergencia hacer
7      δ ← p^T q
8      α ← σ_1/δ
9      x ← x + αp
10     r ← r − αq
11     σ_2 ← r^T r
12     β ← σ_2/σ_1
13     σ_1 ← σ_2
14     p ← r + βp
15     q ← Ap
16 fin para
```

la secuencia de residuos generada por el método pierde progresivamente la ortogonalidad, con frecuencia la acaba perdiendo por completo, y esta secuencia puede incluso volverse linealmente dependiente. En dichos casos, la aproximación generada por el método *no* es la aproximación óptima entre todas las del subespacio de Krylov y no está claro hasta qué punto se cumplen los resultados teóricos del apartado anterior. Por ejemplo, en [80] se presenta un ejemplo numérico en el que, utilizando aritmética de doble precisión, el método *no* calcula una buena aproximación tras n iteraciones (A-norma del error del orden de 10^{-2}), y para el que requiere cerca de $2n$ iteraciones para encontrar una buena aproximación (A-norma del error por debajo de 10^{-12}). Por esta razón, entre otras, el algoritmo no adquirió mucha popularidad hasta que Reid defendió en [133] que, a pesar del efecto de los errores de redondeo, el método es muy efectivo en relación con otros métodos iterativos[1]. Además, el método se aplica a sistemas de ecuaciones de gran dimensión, en los que no es viable computacionalmente llevar a cabo n iteraciones. Como consecuencia de este tipo de observaciones, es común categorizar a este método como puramente iterativo. En este contexto, el análisis de convergencia es útil para cuantificar, teóricamente, la rapidez con la que la secuencia de vectores $x^{(1)}, x^{(2)}, \ldots$ calculada por el método, se aproxima a la solución del sistema. En este apartado revisamos los resultados más importantes referentes a la convergencia del método CG cuando se usa como método iterativo.

Los resultados que exploramos seguidamente se basan en la conexión que existe entre el subespacio de Krylov $\mathcal{K}_i(A, r_0)$ y el conjunto de todos los polinomios de grado inferior a i. Como indicamos en el Apartado 3.1.1, el subespacio de Krylov $\mathcal{K}_i(A, r_0)$ se define como el conjunto de todas las combinaciones lineales de los vectores $r^{(0)}, Ar^{(0)}, \ldots, A^{i-1}r^{(0)}$. Por lo tanto, un miembro genérico de este conjunto tiene la forma

$$c_0 r^{(0)} + c_1 A r^{(0)} + \cdots + c_{j-1} A^{i-1} r^{(0)} = (c_0 I + c_1 A + \cdots + c_{j-1} A^{i-1}) r^{(0)} = q(A) r^{(0)},$$

donde $q(z) = c_0 + c_1 z + \cdots + c_{i-1} z^{i-1}$ es un polinomio de grado inferior a i. Además, para cualquier polinomio q de grado inferior a i, el vector $q(A)r^{(0)} \in \mathcal{K}_i(A, r_0)$. Por lo tanto, si $\mathcal{P}_{i-1}$ denota el conjunto de todos los polinomios de grado inferior a i, entonces se cumple que

$$\mathcal{K}_i(A, r_0) = \{q(A) r^{(0)} : q \in \mathcal{P}_{i-1}\}.$$

[1] Es necesario mencionar que el estado del arte ya dispone de la teoría suficiente para comprender el efecto de los errores de redondeo en el algoritmo de Lanczos (algoritmo equivalente al método CG), incluyendo por qué la secuencia de vectores de Lanczos sigue siendo efectiva para resolver sistemas de ecuaciones lineales a pesar de la pérdida de ortogonalidad; para más detalles consultar [80] y sus referencias.

A partir de esta conexión, es posible identificar una nueva relación entre los vectores de error asociados a los vectores del subespacio $x^{(0)} + \mathcal{K}_i(A, r_0)$ y los polinomios de grado inferior a $i+1$ que toman el valor 1 en el origen. Un miembro arbitrario y de este subespacio tiene la forma $y = x^{(0)} + q(A)r^{(0)}$, donde $q \in \mathcal{P}_{i-1}$. Por lo tanto, el error asociado $e_y = y - x$ cumple que

$$e_y = x^{(0)} + q(A)r^{(0)} - x = e^{(0)} - q(A)Ae^{(0)} = (I - q(A)A)e^{(0)} = p(A)e^{(0)},$$

donde $p(z) = 1 - q(z)z$ es un polinomio $p \in \mathcal{P}_i$, con $p(0) = 1$. En sentido contrario, cualquier polinomio $p \in \mathcal{P}_j$, con $p(0) = 1$, tiene la forma $p(z) = 1 - q(z)z$, con $q \in \mathcal{P}_{j-1}$. Por lo tanto, $p(A)e^{(0)} = x - y$, para algún $y \in x^{(0)} + \mathcal{K}_i(A, r_0)$. Esta conexión entre los vectores de error y el conjunto de los polinomios con grado inferior a $i+1$ con valor 1 en el origen, permite reescribir (3.16) como

$$\|e^{(i)}\|_A = \min_{p \in \mathcal{P}_i \wedge p(0)=1} \|p(A)e^{(0)}\|_A. \tag{3.28}$$

La expresión (3.28) es útil para derivar *una cota superior de* $\|e^{(i)}\|_A$ *en términos de los valores propios de* A. De aquí en adelante, asumimos que $\{v_1, v_2, \ldots, v_n\}$, con $v_i \in \mathbb{R}^n$, es una base ortonormal de vectores propios de A; es decir, un conjunto de vectores propios de A para los cuales se cumple que

$$v_l^T v_k = \begin{cases} 1 & si \quad l = k, \\ 0 & si \quad l \neq k; \end{cases}$$

además, denotamos por $\lambda_i \in \mathbb{R}$ el valor propio asociado al vector propio v_i, es decir, $Av_i = \lambda_i v_i$. Un vector propio v_i y su valor propio asociado λ_i cumplen para cualquier polinomio arbitrario q, que

$$q(A)v_i = q(\lambda_i)v_i.$$

Esta relación resultará de gran utilidad para transformar las expresiones que siguen.

Considerando el vector $e^{(0)}$ como una combinación lineal de $\{v_1, v_2, \ldots, v_n\}$

$$e^{(0)} = \sum_{j=1}^{n} \epsilon_j v_j,$$

se cumple que

$$\begin{aligned} \|p(A)e^{(0)}\|_A^2 &= (p(A)e^{(0)})^T A(p(A)e^{(0)}) = \left(\sum_{j=1}^{n} \epsilon_j (p(A)v_j)^T\right) A \left(\sum_{j=1}^{n} \epsilon_j p(A)v_j\right) \\ &= \left(\sum_{j=1}^{n} \epsilon_j (p(\lambda_j)v_j)^T\right) A \left(\sum_{j=1}^{n} \epsilon_j p(\lambda_j)v_j\right) = \sum_{l=1}^{n}\sum_{k=1}^{n} \epsilon_l \epsilon_k p(\lambda_l)p(\lambda_k) v_l^T A v_k \\ &= \sum_{k=1}^{n}\sum_{l=1}^{n} \epsilon_k \epsilon_l p(\lambda_l)\lambda_k p(\lambda_k) v_l v_k^T = \sum_{k=1}^{n} \epsilon_k^2 \lambda_k [p(\lambda_k)]^2. \end{aligned}$$

Combinando esta expresión con (3.28), obtenemos que

$$\|e^{(i)}\|_A^2 = \min_{p \in \mathcal{P}_i \wedge p(0)=1} \sum_{k=1}^{n} \epsilon_k^2 \lambda_k [p(\lambda_k)]^2. \tag{3.29}$$

Para obtener un resultado que no haga referencia a los coeficientes de la expansión de $e^{(0)}$, podemos tener cuenta que

$$\|e^{(0)}\|_A^2 = (e^{(0)})^T A e^{(0)} = \sum_{k=1}^{n} \epsilon_k^2 \lambda_k,$$

y, por lo tanto,

$$\sum_{k=1}^{n} \epsilon_k^2 \lambda_k [p(\lambda_k)]^2 \leq \max_{k=1,\ldots,n} [p(\lambda_k)]^2 \|e^{(0)}\|_A^2.$$

Por último, combinando esta expresión con (3.29), obtenemos una cota superior para el error relativo

$$\frac{\|e^{(i)}\|_A}{\|e^{(0)}\|_A} \leq \min_{p \in \mathcal{P}_i \wedge p(0)=1} \max_{k=1,\ldots,n} |p(\lambda_k)|. \tag{3.30}$$

Partiendo de esta cota superior, se puede estimar la magnitud del error relativo tras i iteraciones, encontrando aquel polinomio de grado inferior a $i+1$, con valor 1 en el origen, más cercano a cero en el conjunto de valores propios de la matriz A. En caso de que la matriz A sólo tenga m valores propios distintos, entonces se puede encontrar fácilmente un polinomio para el que la cota superior (3.30) toma el valor cero (es decir, $x^{(m)}$ es la solución del sistema). Si denotamos $\bar{\lambda}_1, \ldots, \bar{\lambda}_m$ como los valores propios distintos de A, y escogemos el polinomio de grado m, $p(z) = c(z-\bar{\lambda}_1)(z-\bar{\lambda}_2)\cdots(z-\bar{\lambda}_m)$, donde c se selecciona para que $p(0)=1$, entonces $p(\lambda_k)=0$, para $k=1,\ldots,n$ y, por (3.30), $\|e^{(m)}\|_A/\|e^{(0)}\|_A$ es cero. De forma más general, podemos considerar una matriz A cuyos valores propios están situados en m *clusters*. Si escogemos el polinomio de grado m, $p(z) = c(z-\hat{\lambda}_1)(z-\hat{\lambda}_2)\cdots(z-\hat{\lambda}_m)$, donde $\hat{\lambda}_1, \ldots, \hat{\lambda}_m$ son valores representativos de cada uno de los m clusters, entonces $\max_{k=1,\ldots,n}|p(\lambda_k)|$ será pequeño, ya que todo valor propio, λ_k, está cerca del representante de un cluster, $\hat{\lambda}_s$, y para todo $\hat{\lambda}_s$, $p(\hat{\lambda}_s)=0$. Por lo tanto, por (3.30), se puede concluir que si el espectro de A está concentrado en m clusters, con $m \ll n$, entonces el algoritmo convergerá rápidamente a la solución del sistema. Esta conclusión también se puede extender a un espectro en el que la mayoría de los valores propios se concentran en un punto, pero en el que hay un número reducido de valores propios dispersados, ya que cada uno de estos puntos puede considerarse como un *cluster* con un único valor propio.

En la práctica, resulta útil disponer de una cota para el error relativo que sólo dependa de los valores propios *extremos* que, con frecuencia, se pueden calcular económicamente. Si definimos el conjunto de valores propios $\lambda_1, \ldots, \lambda_n$ de tal forma que $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, los valores propios extremos son λ_1 y λ_n. Para obtener esta cota, se escoge un polinomio $p(z)$ que minimice (3.30) en el intervalo $[\lambda_1, \lambda_n]$, en lugar de considerar un conjunto finito de puntos, $\lambda_1, \ldots, \lambda_n$. Los polinomios que cumplen esta propiedad se construyen a partir de los polinomios de Chebyshev. En [138] puede encontrarse una descripción de las propiedades de este tipo de polinomios. Para este trabajo, resulta interesante que, usando los polinomios de Chebyshev, se puede derivar la siguiente cota superior para el error relativo:

$$\frac{\|e^{(i)}\|_A}{\|e^{(0)}\|_A} \leq 2\left(\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}\right)^i,$$

donde $\kappa(A) = \lambda_n/\lambda_1$ es el número de condición espectral de la matriz A. A pesar de que esta cota suele ser pesimista, proporciona información útil sobre el rendimiento del método CG; en concreto, la velocidad de convergencia es, en el peor de los casos, lineal, con *ratio* de convergencia

$$\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1},$$

y, además, el rendimiento del método puede mejorarse usando un precondicionador que reduzca el número de condición de la matriz, como se describe en el siguiente apartado.

3.1.3. Precondicionado

El método de los Gradientes Conjugados Precondicionados (PCG) se deriva aplicando el método CG al sistema transformado

$$\hat{A}\hat{x} = \hat{b} \;\equiv\; (L^{-1}AL^{-T})(L^T x) = (L^{-1}b), \tag{3.31}$$

donde L es el factor de Cholesky de un precondicionador SDP dado, M; es decir, $M = LL^T$. Si $\kappa(L^{-1}AL^{-T}) \ll \kappa(A)$, o los valores propios de $L^{-1}AL^{-T}$ están "mejor agrupados" que los de A, en el sentido discutido en el Apartado 3.1.2, entonces el método PCG converge más rápido a la solución del sistema (3.31) que el método CG al resolver el sistema de partida.

Si aplicamos el Algoritmo 3.1 al sistema (3.31), obtenemos el Algoritmo 3.3. En este algoritmo, el vector $\hat{x}^{(i)}$ es una aproximación de $\hat{x}$, y $\hat{r}^{(i)}$ es el residuo en el espacio de coordenadas del sistema transformado, es decir, $\hat{r}^{(i)} = \hat{b} - \hat{A}\hat{x}^{(i)}$. Si el Algoritmo 3.3 converge tras k iteraciones, la aproximación del sistema original se puede obtener resolviendo el sistema lineal de ecuaciones $x^{(k)} = L^{-T}\hat{x}^{(k)}$.

Algoritmo 3.3: Método CG aplicado al sistema (3.31).

1 Fijar una solución inicial $\hat{x}^{(0)}$
2 $\hat{r}^{(0)} \leftarrow \hat{b} - \hat{A}\hat{x}^{(0)}$
3 $\hat{p}^{(0)} \leftarrow \hat{r}^{(0)}$
4 **para** $i = 0, 1, \ldots,$ *hasta convergencia* **hacer**
5 $\quad \alpha^{(i)} \leftarrow \frac{(\hat{r}^{(i)})^T\hat{r}^{(i)}}{(\hat{p}^{(i)})^T L^{-1}AL^{-T}\hat{p}^{(i)}}$
6 $\quad \hat{x}^{(i+1)} \leftarrow \hat{x}^{(i)} + \alpha^{(i)}\hat{p}^{(i)}$
7 $\quad \hat{r}^{(i+1)} \leftarrow \hat{r}^{(i)} - \alpha^{(i)}L^{-1}AL^{-T}\hat{p}^{(i)}$
8 $\quad \beta^{(i+1)} \leftarrow \frac{(\hat{r}^{(i+1)})^T\hat{r}^{(i+1)}}{(\hat{r}^{(i)})^T\hat{r}^{(i)}}$
9 $\quad \hat{p}^{(i+1)} \leftarrow \hat{r}^{(i+1)} + \beta^{(i+1)}\hat{p}^{(i)}$
10 **fin para**

El Algoritmo 3.3 tiene el inconveniente de que el factor de Cholesky L del precondicionador se debe calcular de antemano. Sin embargo, este inconveniente se puede resolver aplicando un cambio de coordenadas. En concreto, definiendo $\hat{x}^{(i)} = L^T x^{(i)}$, $\hat{p}^{(i)} = L^T p^{(i)}$ y $\hat{r}^{(i)} = L^{-1}r^{(i)}$, el Algoritmo 3.3 se transforma en el Algoritmo 3.4. Multiplicando las expresiones de las líneas 2 y 7 de este último algoritmo por L, las de las líneas 3, 6, y 9 por L^{-T} y, por último, definiendo un nuevo vector, $z^{(i)}$, como la solución del sistema de ecuaciones lineales, $Mz^{(i)} = r^{(i)}$, entonces el método PCG queda finalmente como se muestra en el Algoritmo 3.5. En este algoritmo, el vector $x^{(i)}$ es una aproximación de x, y $r^{(i)}$ es el residuo en el espacio de coordenadas del sistema original, es decir, $r^{(i)} = b - Ax^{(i)}$. Además, éste ya no hace referencia al factor de Cholesky L, sino que, en su lugar, el requisito es poder resolver sistemas de ecuaciones lineales, $Mz^{(i)} = r^{(i)}$. Esta operación recibe el nombre de *aplicación del precondicionador*.

Algoritmo 3.4: Método CG aplicado al sistema (3.31) con cambio de coordenadas.

1 Fijar una solución inicial $x^{(0)}$
2 $L^{-1}r^{(0)} \leftarrow L^{-1}b - L^{-1}AL^{-T}L^T x^{(0)}$
3 $L^T p^{(0)} \leftarrow L^{-1}r^{(0)}$
4 **para** $i = 0, 1, \ldots,$ *hasta convergencia* **hacer**
5 $\quad \alpha^{(i)} \leftarrow \frac{(L^{-1}r^{(i)})^T L^{-1}r^{(i)}}{(L^T p^{(i)})^T L^{-1}AL^{-T}Lp^{(i)}}$
6 $\quad L^T x^{(i+1)} \leftarrow L^T x^{(i)} + \alpha^{(i)}L^T p^{(i)}$
7 $\quad L^{-1}r^{(i+1)} \leftarrow L^{-1}r^{(i)} - \alpha^{(i)}L^{-1}AL^{-T}L^T p^{(i)}$
8 $\quad \beta^{(i+1)} \leftarrow \frac{(L^{-1}r^{(i+1)})^T L^{-1}r^{(i+1)}}{(L^{-1}r^{(i)})^T L^{-1}r^{(i)}}$
9 $\quad L^T p^{(i+1)} \leftarrow L^{-1}r^{(i+1)} + \beta^{(i+1)}L^T p^{(i+1)}$
10 **fin para**

Algoritmo 3.5: Método de los gradientes conjugados precondicionados-PCG.

1 Fijar una solución inicial $x^{(0)}$
2 $r^{(0)} \leftarrow b - Ax^{(0)}$
3 Resolver $Mz^{(0)} = r^{(0)}$
4 $p^{(0)} \leftarrow z^{(0)}$
5 **para** $i = 0, 1, \ldots,$ *hasta convergencia* **hacer**
6 $\quad \alpha^{(i)} \leftarrow \frac{(r^{(i)})^T z^{(i)}}{(p^{(i)})^T A p^{(i)}}$
7 $\quad x^{(i+1)} \leftarrow x^{(i)} + \alpha^{(i)} p^{(i)}$
8 $\quad r^{(i+1)} \leftarrow r^{(i)} - \alpha^{(i)} A p^{(i)}$
9 $\quad$ Resolver $Mz^{(i+1)} = r^{(i+1)}$
10 $\quad \beta^{(i+1)} \leftarrow \frac{(r^{(i+1)})^T z^{(i+1)}}{(r^{(i)})^T z^{(i)}}$
11 $\quad p^{(i+1)} \leftarrow z^{(i+1)} + \beta^{(i+1)} p^{(i+1)}$
12 **fin para**

3.2. Aplicación de precondicionadores LDL^T multinivel

3.2.1. Derivación del algoritmo

El precondicionador multinivel $M \approx A$, obtenido haciendo uso del Algoritmo 2.15, se puede expresar recursivamente como

$$M = D^{-1}\tilde{P}^{-T}P^{-T}\begin{pmatrix}\tilde{L}_B & 0\\ \tilde{L}_F & I\end{pmatrix}\begin{pmatrix}\tilde{D}_B & 0\\ 0 & M_{\hat{S}_C}\end{pmatrix}\begin{pmatrix}\tilde{L}_B^T & \tilde{L}_F^T\\ 0 & I\end{pmatrix}P^T\tilde{P}^T D^{-1},$$

donde D es la matriz de escalado del paso 1; $\tilde{P}$ y P son las matrices de permutación de los pasos 2 y 3, respectivamente; $\tilde{L}_B$, $\tilde{L}_F$ y $\tilde{D}_B$ son los bloques de los factores de (2.77); y $M_{\hat{S}_C} \approx \hat{S}_C$ es el precondicionador del complemento de Schur aproximado, $\hat{S}_C$, construido en niveles posteriores del Algoritmo 2.15. Por lo tanto, el algoritmo que aplica este precondicionador se obtiene considerando el proceso de resolución del siguiente sistema de ecuaciones lineales

$$\begin{pmatrix}\tilde{L}_B & 0\\ \tilde{L}_F & I\end{pmatrix}\begin{pmatrix}\tilde{D}_B & 0\\ 0 & M_{\hat{S}_C}\end{pmatrix}\begin{pmatrix}\tilde{L}_B^T & \tilde{L}_F^T\\ 0 & I\end{pmatrix}P^T\tilde{P}^T D^{-1}z = P^T\tilde{P}^T Dr. \tag{3.32}$$

La matriz de escalado que se aplica en el paso 1 del Algoritmo 2.15 transforma el vector residuo r y el vector residuo precondicionado z como

$$r \to Dr = \tilde{r} \quad \text{y} \quad z \to D^{-1}z = \tilde{z}, \tag{3.33}$$

y la matriz de permutación que se aplica en el paso 2 transforma $\tilde{r}$ y $\tilde{z}$ como

$$\tilde{r} \to \tilde{P}^T\tilde{r} = \hat{r} \quad \text{y} \quad \tilde{z} \to \tilde{P}\tilde{z} = \hat{z}. \tag{3.34}$$

Aplicando la matriz de permutación del paso 3 sobre $\hat{r}$ y $\hat{z}$, obtenemos

$$\hat{r} \to P^T\hat{r} = \bar{r} \quad \text{y} \quad \hat{z} \to P\hat{z} = \bar{z}. \tag{3.35}$$

Particionando $\bar{r}$ y $\bar{z}$ conforme a (2.77)

$$\bar{r} \equiv \begin{pmatrix}\bar{r}_B\\ \bar{r}_C\end{pmatrix} \quad \text{y} \quad \bar{z} \equiv \begin{pmatrix}\bar{z}_B\\ \bar{z}_C\end{pmatrix}, \tag{3.36}$$

el sistema (3.32) queda finalmente como

$$\begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & M_{\hat{S}_C} \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} = \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix}.$$

La solución de este sistema se obtiene resolviendo, en primer lugar, un sistema triangular inferior unidad de la forma

$$\begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix};$$

a continuación, debe obtenerse "recursivamente" la solución del sistema

$$\begin{pmatrix} \tilde{D}_B & 0 \\ 0 & M_{\hat{S}_C} \end{pmatrix} \begin{pmatrix} \tilde{y}_B \\ \tilde{y}_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix};$$

y, por último, es necesario resolver un sistema triangular superior unidad de la forma

$$\begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} = \begin{pmatrix} \tilde{y}_B \\ \tilde{y}_C \end{pmatrix}.$$

La aplicación del precondicionador LDL^T multinivel se muestra, *en su forma recursiva*, en el Algoritmo 3.6. Este algoritmo recibe como entrada el vector residuo, r, y devuelve como resultado el residuo precondicionado, z (ver líneas 3 y 9 del Algoritmo 3.5). El *caso recursivo* del Algoritmo 3.6 (ver paso 5) resuelve el sistema $M_{\hat{S}_C}\tilde{y}_C = y_C$, donde $M_{\hat{S}_C} \approx \hat{S}_C$ es el precondicionador construido para el sistema complemento de Schur aproximado en niveles posteriores del Algoritmo 2.15. Dependiendo de la estructura de la última factorización *aproximada* calculada por este último algoritmo, podemos distinguir *dos casos base* para la recursión del Algoritmo 3.6:

- Si esta factorización es de la forma

$$\hat{A} = B = \tilde{L}_B \tilde{D}_B \tilde{L}_B^T + R_B, \tag{3.37}$$

  entonces los bloques que aparecen en el Algoritmo 3.6 con subíndice F o C son *vacíos*. Por tanto, el primer caso base define $\tilde{y}_C$ como un bloque vacío (ver paso 6).

- Si esta factorización es de la forma

$$P^T \hat{A} P = \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} + R,$$

  entonces los bloques que aparecen en el Algoritmo 3.6 con subíndice F o C no están vacíos y, además, $M_{\hat{S}_C} = \hat{S}_C = L_C D_C L_C^T$. Por lo tanto, el segundo caso base resuelve *directamente* el sistema complemento de Schur aproximado $\hat{S}_C \tilde{y}_C = L_C D_C L_C^T \tilde{y}_C = y_C$ (ver paso 6).

Para facilitar la exposición, consideramos, además, la *forma iterativa* del proceso de aplicación del precondicionador multinivel, que a su vez descomponemos en dos partes: resolución progresiva y regresiva. La resolución progresiva se recoge en el Algoritmo 3.7, y la regresiva en el Algoritmo 3.8. El primer algoritmo parte del primer nivel construido por el Algoritmo 2.15, y procede hasta el último repitiendo los pasos 1, 2, 3 y 4. Por otro lado, el segundo parte del último nivel y procede en sentido inverso hasta el primero repitiendo los pasos 7, 8, 9, y 10. Los pasos 5 y 11 de los Algoritmos 3.7 y 3.8 se obtienen a partir de la llamada recursiva del paso 5 del Algoritmo 3.6. En concreto, el paso 5 del Algoritmo 3.7 proporciona el vector y_C obtenido en el nivel actual como entrada al nivel siguiente, $r \equiv y_C$, y el paso 11 del Algoritmo 3.8 proporciona la salida del nivel actual, z, al nivel previo, $\tilde{y}_C \equiv z$.

Algoritmo 3.6: Resuelve $Mz = r$, siendo $M \approx A$ el precondicionador construido por el Algoritmo 2.15.

1. Aplicar la matriz de escalado diagonal D, $r \to Dr = \tilde{r}$
2. Aplicar la matriz de permutación $\hat{P}^T$, $\tilde{r} \to \hat{P}^T\tilde{r} = \hat{r}$
3. Aplicar la matriz de permutación P^T y reparticionar
$$\hat{r} \to P^T\hat{r} \equiv \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix}$$
4. Resolver el sistema triangular inferior unidad
$$\begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix} \tag{3.38}$$
si *(3.38) no se corresponde con la última fact. aproximada calculada por el Algoritmo 2.15*
5. Resolver $Mz = r$, mediante el Algoritmo 3.6 con $M \equiv M_{\hat{S}_C}$, $z \equiv \tilde{y}_C$, y $r \equiv y_C$

si no
6. Si y_C es un bloque vacío, entonces definir $\tilde{y}_C$ como un bloque vacío
Si no, resolver $L_C D_C L_C^T \tilde{y}_C = y_C$

fin si
7. Calcular $\tilde{y}_B \leftarrow \tilde{D}_B^{-1} y_B$ y resolver el sistema triangular superior unidad
$$\begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} = \begin{pmatrix} \tilde{y}_B \\ \tilde{y}_C \end{pmatrix}$$
8. Aplicar la matriz de permutación P y reparticionar
$$\begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} \to P \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} \equiv \hat{z}$$
9. Aplicar la matriz de permutación $\hat{P}$, $\hat{z} \to \hat{P}\hat{z} = \tilde{z}$
10. Aplicar la matriz de escalado diagonal D, $\tilde{z} \to D\tilde{z} = z$

3.2.2. Detalles de implementación de la biblioteca ILUPACK

Como ya se introdujo en el Apartado 2.2.3, el paso 3 del Algoritmo 2.15 descarta el bloque $\tilde{L}_F$ de la factorización parcial (2.77), guardando, en su lugar, el bloque F de $P^T\hat{A}P$. Esta estrategia, que se usa en la biblioteca con el objetivo de ahorrar memoria, conlleva cambios en el Algoritmo 3.6. En concreto, estos cambios atañen a los pasos 4 y 7 que, para facilitar la exposición, descomponemos respectivamente en

$$\text{Resolver } \tilde{L}_B y_B = \bar{r}_B \tag{3.41}$$
$$y_C \leftarrow \bar{r}_C - L_F y_B, \tag{3.42}$$

y

$$\tilde{y}_B \leftarrow \tilde{D}_B^{-1} y_B \quad ; \quad \bar{z}_C \leftarrow \tilde{y}_C \tag{3.43}$$
$$\text{Resolver } \tilde{L}_B^T \bar{z}_B = \tilde{y}_B - \tilde{L}_F^T \bar{z}_C. \tag{3.44}$$

Algoritmo 3.7: Resuelve $Mz = r$. Parte I: resolución progresiva.

1. Aplicar la matriz de escalado diagonal D, $r \to Dr = \tilde{r}$
2. Aplicar la matriz de permutación $\hat{P}^T$, $\tilde{r} \to \hat{P}^T\tilde{r} = \hat{r}$
3. Aplicar la matriz de permutación P^T y reparticionar
$$\hat{r} \to P^T\hat{r} \equiv \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix}$$
4. Resolver el sistema triangular inferior unidad
$$\begin{pmatrix} \tilde{L}_B & 0 \\ \tilde{L}_F & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix} \tag{3.39}$$
5. Proceder con el siguiente nivel repitiendo los pasos 1, 2, 3 y 4 con $r \equiv y_C$, hasta que finalmente (3.39) se corresponda con la última factorización *aproximada* calculada por el Algoritmo 2.15

Algoritmo 3.8: Resuelve $Mz = r$. Parte II: resolución regresiva.

6. Si y_C es un bloque vacío, definir $\tilde{y}_C$ como un bloque vacío
En caso contrario, resolver $L_C D_C L_C^T \tilde{y}_C = y_C$ y proceder con el nivel anterior
7. Calcular $\tilde{y}_B \leftarrow \tilde{D}_B^{-1} y_B$ y resolver el sistema triangular superior unidad
$$\begin{pmatrix} \tilde{L}_B^T & \tilde{L}_F^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} = \begin{pmatrix} \tilde{y}_B \\ \tilde{y}_C \end{pmatrix} \tag{3.40}$$
8. Aplicar la matriz de permutación P y reparticionar
$$\begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} \to P \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix} \equiv \hat{z}$$
9. Aplicar la matriz de permutación $\hat{P}$, $\hat{z} \to \hat{P}\hat{z} = \tilde{z}$
10. Aplicar la matriz de escalado diagonal D, $\tilde{z} \to D\tilde{z} = z$
11. Proceder con el nivel anterior repitiendo los pasos 7, 8, 9 y 10, con
$$\tilde{y}_C \equiv z$$
hasta que (3.40) se corresponda con la primera factorización *aproximada* calculada por el Algoritmo 2.15

Los cambios en las expresiones (3.41)-(3.44) se derivan teniendo en cuenta que, al descartar $\tilde{L}_F$ y guardar F en su lugar, se obtiene la siguiente factorización aproximada por bloques

$$P^T \hat{A} P = \begin{pmatrix} B & F^T \\ F & C \end{pmatrix} = \begin{pmatrix} \tilde{L}_B & 0 \\ F\tilde{L}_B^{-T}\tilde{D}_B^{-1} & I \end{pmatrix} \begin{pmatrix} \tilde{D}_B & 0 \\ 0 & \hat{S}_C \end{pmatrix} \begin{pmatrix} \tilde{L}_B^T & \tilde{D}_B^{-1}\tilde{L}_B^{-1}F^T \\ 0 & I \end{pmatrix} + R.$$

Por lo tanto, la expresión (3.42) se transforma en

$$y_C \leftarrow \bar{r}_C - F\tilde{L}_B^{-T}\tilde{D}_B^{-1}y_B, \tag{3.45}$$

y la (3.44) en

$$\text{Resolver } \tilde{L}_B^T \bar{z}_B = \tilde{D}_B^{-1}y_B - \tilde{D}_B^{-1}\tilde{L}_B^{-1}F^T\bar{z}_C. \tag{3.46}$$

Combinando la expresión (3.41) con (3.45), obtenemos que

$$y_C \leftarrow \bar{r}_C - F\tilde{L}_B^{-T}\tilde{D}_B^{-1}\tilde{L}_B^{-1}\bar{r}_B, \tag{3.47}$$

y definiendo la siguiente operación, en substitución de (3.41),

$$\text{Resolver } \tilde{L}_B\tilde{D}_B\tilde{L}_B^T s_B = \bar{r}_B,$$

entonces (3.45) se transforma en

$$y_C \leftarrow \bar{r}_C - Fs_B, \tag{3.48}$$

y (3.46), en

$$\text{Resolver } \tilde{L}_B\tilde{D}_B\tilde{L}_B^T\bar{s}_B = F^T\bar{z}_C \quad ; \quad \bar{z}_B \leftarrow s_B - \bar{s}_B.$$

El Algoritmo 3.9 tiene en cuenta las consideraciones anteriores para aplicar el precondicionador LDL^T multinivel conforme a la biblioteca ILUPACK. La implementación de este algoritmo en la biblioteca se basa en dos núcleos computacionales que calculan dos operaciones *matriciales* básicas: (1) multiplicación de una matriz dispersa por un vector; y (2) aplicación de una factorización LDL^T. Las operaciones del primer tipo son de la forma "$r \leftarrow Fz$" o "$r \leftarrow F^Tz$", donde $F \in \mathbb{R}^{m,n}$ es una matriz rectangular dispersa, y $r \in \mathbb{R}^m$ y $z \in \mathbb{R}^n$ son vectores; ver líneas 5 y 7. Las operaciones del segundo tipo consisten en la resolución de sistemas de la forma $LDL^Tz = r$, donde $L \in \mathbb{R}^{n,n}$ es una matriz triangular inferior unidad dispersa y $D \in \mathbb{R}^{n,n}$ es una matriz diagonal; ver líneas 4 y 5. En comparación con el Algoritmo 3.6, este algoritmo multiplica los bloques F y F^T por los vectores s_B y $\bar{z}_C$, en lugar de los bloques $\tilde{L}_F$ y $\tilde{L}_F^T$ por y_B y $\bar{z}_C$. El bloque $\tilde{L}_F$ tiene habitualmente un mayor número de elementos no nulos que F y, por tanto, la multiplicación de F por un vector requiere con frecuencia un menor número de operaciones aritméticas que la multiplicación de $\tilde{L}_F$ por un vector. Sin embargo, en cada nivel del Algoritmo 3.9 se resuelven dos sistemas triangulares adicionales, inferior y superior, y se aplica una vez más la matriz $\tilde{D}_B^{-1}$.

3.3. Aplicación paralela de precondicionadores LDL^T multinivel

3.3.1. Aplicación del precondicionador transformado. Análisis de concurrencia

El procedimiento que aplica los precondicionadores construidos por el Algoritmo 2.19 se muestra, en su forma iterativa, en los Algoritmos 3.10 (resolución progresiva) y 3.11 (resolución regresiva). El objetivo de este apartado es analizar la concurrencia que se puede explotar en estos algoritmos, para lo cual presentamos paulatinamente las computaciones que éstos realizan para aplicar el precondicionador que se construyó paso a paso partiendo del ejemplo de las Figuras 2.10 y 2.11; ver Apartado 2.3.2.

Algoritmo 3.9: Resuelve $Mz = r$ conforme a la biblioteca ILUPACK.

1 $\tilde{r} \leftarrow Dr$
2 $\hat{r} \leftarrow \tilde{P}^T \tilde{r}$
3 $\begin{pmatrix} \bar{r}_B \\ \bar{r}_C \end{pmatrix} \leftarrow P^T \hat{r}$
4 Resolver $\tilde{L}_B \tilde{D}_B \tilde{L}_B^T s_B = \bar{r}_B$
5 $y_C \leftarrow \bar{r}_C - F s_B$
6 Resolver $Mz = r$, mediante el Algoritmo 3.9 con $M \equiv M_{\hat{S}_C}$, $z \equiv \bar{z}_C$, y $r \equiv y_C$
7 Resolver $\tilde{L}_B \tilde{D}_B \tilde{L}_B^T \bar{s}_B = F^T \bar{z}_C$
8 $\bar{z}_B \leftarrow s_B - \bar{s}_B$
9 $\hat{z} \leftarrow P \begin{pmatrix} \bar{z}_B \\ \bar{z}_C \end{pmatrix}$
10 $\tilde{z} \leftarrow \hat{P} \hat{z}$
11 $z \leftarrow D\tilde{z}$

El Algoritmo 3.10 resuelve el sistema $Mz = r$, donde r es el vector residuo en el espacio del sistema original, $Ax = b$, y z el vector residuo precondicionado. La notación que se usa para describir la **Entrada** del Algoritmo 3.10

$$r \equiv \Omega^T r \equiv \begin{pmatrix} r_X \\ \hline r_Z \end{pmatrix},$$

indica, por una parte, que $r \equiv \Omega^T r$ es el vector residuo de entrada a la primera fase y, por otra, que este vector se puede reparticionar en los bloques r_X y r_Z, donde r_X contiene las componentes correspondientes a la primera fase, y r_Z las componentes correspondientes a fases posteriores. Además, la notación que se usa para describir la **Salida** del Algoritmo 3.10

$$z \equiv \Omega z, \tag{3.51}$$

indica que el vector z que aparece a la izquierda en esta expresión es la solución del sistema $Mz = r$, y se obtiene aplicando la matriz de permutación Ω a z, donde z en este segundo caso se refiere al residuo precondicionado que se obtiene al aplicar todo el precondicionador (ver paso 11 del Algoritmo 3.11).

Para el caso de estudio que nos ocupa, los vectores $r \equiv \Omega^T r$ y z se pueden particionar conforme a la matriz de entrada $A \equiv \Omega^T A \Omega$ del Algoritmo 2.19 (ver (2.87)), obteniéndose

$$r \equiv \Omega^T r = \begin{pmatrix} r_1^{(3,1)} \\ r_2^{(3,2)} \\ r_3^{(3,3)} \\ r_4^{(3,4)} \\ \hline r_5^{(2,1)} \\ r_6^{(2,2)} \\ \hline\hline r_7^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} r_X \\ \hline r_Z \end{pmatrix}, \quad z = \begin{pmatrix} z_1^{(3,1)} \\ z_2^{(3,2)} \\ z_3^{(3,3)} \\ z_4^{(3,4)} \\ \hline z_5^{(2,1)} \\ z_6^{(2,2)} \\ \hline\hline z_7^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} z_X \\ \hline z_Z \end{pmatrix}, \tag{3.52}$$

donde r_X y z_X contienen las componentes de r y z asociadas a los subgrafos independientes (primera fase), y r_Z y z_Z las asociadas a los separadores del segundo y primer nivel (segunda y tercera fase).

Para facilitar la legibilidad del apartado, no consideraremos en detalle los pasos correspondientes a la aplicación de la matriz de escalado diagonal y de la permutación para reducción de llenado; ver pasos 1, 2, 10 y 11 de los algoritmos 3.10 y 3.11. En todos estos pasos se puede

Algoritmo 3.10: Resuelve $Mz = r$. Parte I: resolución progresiva.

Entrada: $r \equiv \Omega^T r \equiv \left(\begin{array}{c} r_X \\ \hline r_Z \end{array}\right)$

Salida : $z \equiv \Omega z$

para *fase* $= 1, \ldots, l$ **hacer**

1. Aplicar la matriz de escalado diagonal,

$$\left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} r_X \\ \hline r_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} r_X \\ \hline r_Z \end{array}\right) = \left(\begin{array}{c} \tilde{r}_X \\ \hline \tilde{r}_Z \end{array}\right)$$

2. Aplicar la matriz de permutación,

$$\left(\begin{array}{c|c} \hat{P}_X^T & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} \tilde{r}_X \\ \hline \tilde{r}_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} \hat{P}_X^T & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \tilde{r}_X \\ \hline \tilde{r}_Z \end{array}\right) = \left(\begin{array}{c} \hat{r}_X \\ \hline \hat{r}_Z \end{array}\right)$$

3. Aplicar la matriz de permutación y reparticionar,

$$\left(\begin{array}{c|c} P_X^T & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} \hat{r}_X \\ \hline \hat{r}_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} P_X^T & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \hat{r}_X \\ \hline \hat{r}_Z \end{array}\right) \equiv \left(\begin{array}{c} \bar{r}_{B_X} \\ \bar{r}_{C_X} \\ \hline \bar{r}_{C_Z} \end{array}\right)$$

4. Resolver el sistema triangular inferior unidad

$$\left(\begin{array}{cc|c} \tilde{L}_{B_X} & 0 & 0 \\ \tilde{L}_{F_X} & I & 0 \\ \hline \tilde{L}_{F_Y} & 0 & I \end{array}\right) \left(\begin{array}{c} y_{B_X} \\ y_{C_X} \\ \hline y_{C_Z} \end{array}\right) = \left(\begin{array}{c} \bar{r}_{B_X} \\ \bar{r}_{C_X} \\ \hline \bar{r}_{C_Z} \end{array}\right) \tag{3.49}$$

5. Proceder con el siguiente nivel repitiendo los pasos 1, 2, 3 y 4, con

$$r \equiv y_C \equiv \left(\begin{array}{c} r_X \\ \hline r_Z \end{array}\right),$$

hasta que (3.49) se corresponda con la última factorización *aproximada* calculada por el Algoritmo 2.19 en la *fase* actual

si *fase* $< l$ **entonces**

6. Proceder con la siguiente fase con

$$r \equiv \Phi^T y_C \equiv \left(\begin{array}{c} r_X \\ \hline r_Z \end{array}\right)$$

fin si

fin para

Algoritmo 3.11: Resuelve $Mz = r$. Parte II: resolución regresiva.

7. Si y_C no es un bloque vacío, resolver el sistema

$$L_C D_C L_C^T \tilde{y}_C = y_C,$$

y reparticionar

$$\tilde{y}_C \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array}\right).$$

Si no, definir $\tilde{y}_C$ como un bloque vacío

para *fase* $= l, \ldots, 1$ **hacer**

8. Calcular $\tilde{y}_{B_X} \leftarrow \tilde{D}_{B_X}^{-1} y_{B_X}$ y resolver el sistema triangular superior unidad,

$$\left(\begin{array}{cc|c} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T & \tilde{L}_{F_Y}^T \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \bar{z}_{B_X} \\ \bar{z}_{C_X} \\ \hline \bar{z}_{C_Z} \end{array}\right) = \left(\begin{array}{c} \tilde{y}_{B_X} \\ \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array}\right) \tag{3.50}$$

9. Aplicar la matriz de permutación y reparticionar,

$$\left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} \bar{z}_{B_X} \\ \bar{z}_{C_X} \\ \hline \bar{z}_{C_Z} \end{array}\right) \rightarrow \left(\begin{array}{c|c} P_X & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \bar{z}_{B_X} \\ \bar{z}_{C_X} \\ \hline \bar{z}_{C_Z} \end{array}\right) \equiv \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right)$$

10. Aplicar la matriz de permutación,

$$\left(\begin{array}{c|c} \hat{P}_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} \hat{P}_X & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right) = \left(\begin{array}{c} \tilde{z}_X \\ \hline \tilde{z}_Z \end{array}\right)$$

11. Aplicar la matriz de escalado diagonal,

$$\left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right), \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right) \rightarrow \left(\begin{array}{c|c} D_X & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right) = \left(\begin{array}{c} z_X \\ \hline z_Z \end{array}\right)$$

12. Proceder con el nivel anterior repitiendo los pasos 8, 9, 10 y 11, con

$$\tilde{y}_C \equiv z \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array}\right),$$

hasta que (3.50) se corresponda con la primera factorización *aproximada* calculada por el Algoritmo 2.19 en la *fase* actual.

si *fase* > 1 **entonces**

13. Proceder con la fase anterior con

$$\tilde{y}_C \equiv \Phi z \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array}\right).$$

fin si

fin para

explotar trivialmente la concurrencia, dado que D_X y $\hat{P}_X$ son matrices diagonales por bloques, que dividen la aplicación en tantos subproblemas independientes como bloques presenten estas matrices. Además, los vectores $\tilde{r}$ y $\hat{r}$ de los pasos 1 y 2, y los vectores $\tilde{z}$ y $\hat{z}$ de los pasos 10 y 11, se pueden particionar conforme a r y z en (3.52).

Resolución progresiva

La primera fase del Algoritmo 3.10 se inicia con el primer nivel construido por el Algoritmo 2.19, realizando las computaciones de los pasos 1, 2, 3 y 4. La concurrencia que se puede explotar en los pasos 3 y 4 se detalla a continuación:

- El paso 3 aplica la matriz de permutación diag($P_X^T \mid I$) sobre $\hat{r}$ y reparticiona el resultado, $\bar{r}$, conforme a la factorización aproximada por bloques (2.93). Por lo tanto, los bloques $\bar{r}_{B_X}$ y $\bar{r}_{C_X}$ se pueden particionar como

$$\bar{r}_{B_X} = \begin{pmatrix} \bar{r}_{B,1}^{(3,1)} \\ \bar{r}_{B,2}^{(3,2)} \\ \bar{r}_{B,3}^{(3,3)} \\ \bar{r}_{B,4}^{(3,4)} \end{pmatrix}, \quad \bar{r}_{C_X} = \begin{pmatrix} \bar{r}_{C,1}^{(3,1)} \\ \bar{r}_{C,2}^{(3,2)} \\ \bar{r}_{C,3}^{(3,3)} \\ \bar{r}_{C,4}^{(3,4)} \end{pmatrix},$$

  donde $\bar{r}_{B,i}^{(3,i)}$ y $\bar{r}_{C,i}^{(3,i)}$, para $i = 1, \ldots, 4$, contienen, respectivamente, las componentes de $\hat{r}_i^{(3,i)}$ pertenecientes a los subconjuntos fino y grueso. Cada par de bloques $(\bar{r}_{B,i}^{(3,i)}, \bar{r}_{C,i}^{(3,i)})$ se puede obtener independientemente del resto aplicando sobre $\hat{r}_i^{(3,i)}$ la matriz de permutación $(P_{ii}^{(3,i)})^T$ (ver (2.88))

$$(P_{ii}^{(3,i)})^T \hat{r}_i^{(3,1)} \equiv \begin{pmatrix} \bar{r}_{B,1}^{(3,1)} \\ \bar{r}_{C,1}^{(3,1)} \end{pmatrix}. \tag{3.53}$$

  Por otra parte, el bloque $\bar{r}_{C_Z}$ contiene el resto de componentes de $\hat{r}$ pertenecientes al subconjunto grueso, que se obtienen *directamente* a partir de $\hat{r}_Z$

$$\bar{r}_{C_Z} = \begin{pmatrix} \bar{r}_{C,5}^{(2,1)} \\ \bar{r}_{C,6}^{(2,2)} \\ \hline\hline \bar{r}_{C,7}^{(1,1)} \end{pmatrix} = \begin{pmatrix} \hat{r}_5^{(2,1)} \\ \hat{r}_6^{(2,2)} \\ \hline\hline \hat{r}_7^{(1,1)} \end{pmatrix}.$$

- Para analizar el paso 4, descomponemos la resolución de (3.49) en dos operaciones: la resolución del sistema triangular inferior

$$\begin{pmatrix} \tilde{L}_{B_X} & 0 \\ \tilde{L}_{F_X} & I \end{pmatrix} \begin{pmatrix} y_{B_X} \\ y_{C_X} \end{pmatrix} = \begin{pmatrix} \bar{r}_{B_X} \\ \bar{r}_{C_X} \end{pmatrix},$$

  y el cálculo del bloque y_{C_Z}

$$y_{C_Z} = \bar{r}_{C_Z} - \tilde{L}_{F_Y} \bar{r}_{B_X}. \tag{3.54}$$

  Partiendo de la definición de los bloques $\tilde{L}_{B_X}$ y $\tilde{L}_{F_X}$ en (2.89), la primera operación se

puede particionar como

$$\begin{pmatrix} \tilde{L}^{(3,1)}_{B,11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \tilde{L}^{(3,2)}_{B,22} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \tilde{L}^{(3,3)}_{B,33} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \tilde{L}^{(3,4)}_{B,44} & 0 & 0 & 0 & 0 \\ \tilde{L}^{(3,1)}_{F,11} & 0 & 0 & 0 & I & 0 & 0 & 0 \\ 0 & \tilde{L}^{(3,2)}_{F,22} & 0 & 0 & 0 & I & 0 & 0 \\ 0 & 0 & \tilde{L}^{(3,3)}_{F,33} & 0 & 0 & 0 & I & 0 \\ 0 & 0 & 0 & \tilde{L}^{(3,4)}_{F,44} & 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} y^{(3,1)}_{B,1} \\ y^{(3,2)}_{B,2} \\ y^{(3,3)}_{B,3} \\ y^{(3,4)}_{B,4} \\ y^{(3,1)}_{C,1} \\ y^{(3,2)}_{C,2} \\ y^{(3,3)}_{C,3} \\ y^{(3,4)}_{C,4} \end{pmatrix} = \begin{pmatrix} \bar{r}^{(3,1)}_{B,1} \\ \bar{r}^{(3,2)}_{B,2} \\ \bar{r}^{(3,3)}_{B,3} \\ \bar{r}^{(3,4)}_{B,4} \\ \bar{r}^{(3,1)}_{C,1} \\ \bar{r}^{(3,2)}_{C,2} \\ \bar{r}^{(3,3)}_{C,3} \\ \bar{r}^{(3,4)}_{C,4} \end{pmatrix},$$

y, por tanto, cada par de bloques $(y^{(3,i)}_{B,i}, y^{(3,i)}_{C,i})$, para $i = 1, \ldots, 4$, se puede calcular de forma independiente resolviendo el siguiente sistema triangular inferior (ver (2.88)):

$$\begin{pmatrix} \tilde{L}^{(3,i)}_{B,ii} & 0 \\ \tilde{L}^{(3,i)}_{F,ii} & I \end{pmatrix} \begin{pmatrix} y^{(3,i)}_{B,i} \\ y^{(3,i)}_{C,i} \end{pmatrix} = \begin{pmatrix} \bar{r}^{(3,i)}_{B,i} \\ \bar{r}^{(3,i)}_{C,i} \end{pmatrix}. \tag{3.55}$$

La segunda operación se puede particionar como (ver definición de $\tilde{L}_{F_Y}$ en (2.91))

$$\begin{pmatrix} y^{(2,1)}_{C,5} \\ \hline y^{(2,2)}_{C,6} \\ \hline\hline y^{(1,1)}_{C,7} \end{pmatrix} = \begin{pmatrix} \bar{r}^{(2,1)}_{C,5} \\ \hline \bar{r}^{(2,2)}_{C,6} \\ \hline\hline \bar{r}^{(1,1)}_{C,7} \end{pmatrix} - \left(\begin{array}{cccc} \tilde{L}^{(3,1)}_{F,51} & \tilde{L}^{(3,2)}_{F,52} & 0 & 0 \\ 0 & 0 & \tilde{L}^{(3,3)}_{F,63} & \tilde{L}^{(3,4)}_{F,64} \\ \hline\hline \tilde{L}^{(3,1)}_{F,71} & \tilde{L}^{(3,2)}_{F,72} & \tilde{L}^{(3,3)}_{F,73} & \tilde{L}^{(3,4)}_{F,74} \end{array}\right) \begin{pmatrix} \bar{r}^{(3,1)}_{B,1} \\ \bar{r}^{(3,2)}_{B,2} \\ \bar{r}^{(3,3)}_{B,3} \\ \bar{r}^{(3,4)}_{B,4} \end{pmatrix}.$$

Desarrollando esta última expresión, el substraendo se puede descomponer como sigue

$$\begin{aligned} y_{C_Z} = \bar{r}_{C_Z} - & \\ & \left[\begin{pmatrix} \tilde{L}^{(3,1)}_{F,51} \bar{r}^{(3,1)}_{B,1} \\ \hline 0 \\ \hline\hline \tilde{L}^{(3,1)}_{F,71} \bar{r}^{(3,1)}_{B,1} \end{pmatrix} + \begin{pmatrix} \tilde{L}^{(3,2)}_{F,52} \bar{r}^{(3,2)}_{B,2} \\ \hline 0 \\ \hline\hline \tilde{L}^{(3,2)}_{F,72} \bar{r}^{(3,2)}_{B,2} \end{pmatrix} + \right. \\ & \left. \begin{pmatrix} 0 \\ \hline \tilde{L}^{(3,3)}_{F,63} \bar{r}^{(3,3)}_{B,3} \\ \hline\hline \tilde{L}^{(3,3)}_{F,73} \bar{r}^{(3,3)}_{B,3} \end{pmatrix} + \begin{pmatrix} 0 \\ \hline \tilde{L}^{(3,4)}_{F,64} \bar{r}^{(3,4)}_{B,4} \\ \hline\hline \tilde{L}^{(3,4)}_{F,74} \bar{r}^{(3,4)}_{B,4} \end{pmatrix} \right], \end{aligned} \tag{3.56}$$

donde el cálculo de cualquiera de estos sumandos se puede llevar a cabo de forma independiente al resto, aunque la acumulación de éstos en y_{C_Z} requiere una sincronización. Como veremos en el Apartado 3.3.2, el algoritmo paralelo gestiona esta acumulación aplicando una estrategia similar a la que se empleó para construir $\hat{S}_{C_Z}$ en el Apartado 2.3.3.

Posteriormente, se procede con el segundo nivel repitiendo los pasos 1, 2, 3, 4 y 5 con

$$r \equiv y_C = \begin{pmatrix} y^{(3,1)}_{C,1} \\ y^{(3,2)}_{C,2} \\ y^{(3,3)}_{C,3} \\ y^{(3,4)}_{C,4} \\ \hline y^{(2,1)}_{C,5} \\ y^{(2,2)}_{C,6} \\ \hline\hline y^{(1,1)}_{C,7} \end{pmatrix} \equiv \begin{pmatrix} r_X \\ \hline r_Z \end{pmatrix},$$

hasta que finalmente el proceso alcanza el último nivel construido por el Algoritmo 2.19 en la primera fase; ver paso 5. La estructura de los vectores y matrices de estos niveles es equivalente a la de los vectores y matrices del primer nivel y, por tanto, se puede explotar el mismo tipo de concurrencia para los cálculos de los pasos 1, 2, 3 y 4.

Tras efectuar las computaciones de la primera fase, el Algoritmo 3.10 procede, en el paso 6, con la segunda. La ordenación del vector de entrada para la segunda fase debe coincidir con la ordenación de la matriz $A \equiv \Phi^T \hat{S}_C \Phi$ (ver (2.95) y Figura 2.13 (c)) y, por tanto, el paso 6 del Algoritmo 3.10 obtiene esta ordenación aplicando la matriz de permutación Φ^T sobre y_C

$$r \equiv \Phi^T y_C = \Phi^T \left(\begin{array}{c} y_{C,1}^{(3,1)} \\ y_{C,2}^{(3,2)} \\ y_{C,3}^{(3,3)} \\ y_{C,4}^{(3,4)} \\ \hline y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \\ \hline\hline y_{C,7}^{(1,1)} \end{array} \right) = \left(\begin{array}{c} y_{C,1}^{(3,1)} \\ y_{C,2}^{(3,2)} \\ y_{C,5}^{(2,1)} \\ y_{C,3}^{(3,3)} \\ y_{C,4}^{(3,4)} \\ y_{C,6}^{(2,2)} \\ \hline\hline y_{C,7}^{(1,1)} \end{array} \right) \equiv \left(\begin{array}{c} r_5^{(2,1)} \\ r_6^{(2,2)} \\ \hline\hline r_7^{(1,1)} \end{array} \right) \equiv \left(\begin{array}{c} r_X \\ \hline\hline r_Z \end{array} \right), \tag{3.57}$$

siendo

$$r_5^{(2,1)} = \left(\begin{array}{c} y_{C,1}^{(3,1)} \\ y_{C,2}^{(3,2)} \\ y_{C,5}^{(2,1)} \end{array} \right), \; r_6^{(2,2)} = \left(\begin{array}{c} y_{C,3}^{(3,3)} \\ y_{C,4}^{(3,4)} \\ y_{C,6}^{(2,2)} \end{array} \right),$$

donde r_X contiene las componentes correspondientes a la segunda fase (separadores del segundo nivel) y r_Z las correspondientes a la tercera (separador del primer nivel).

Repitiendo el análisis de la primera fase para la segunda, la concurrencia que se puede explotar en los pasos 3 y 4 es la siguiente:

- En el paso 3, el vector $\bar{r}$ se reparticiona conforme a (2.96) y, por tanto, los bloques $\bar{r}_{B_X}$ y $\bar{r}_{C_X}$ tienen la siguiente estructura

$$\bar{r}_{B_X} = \left(\begin{array}{c} \bar{r}_{B,5}^{(2,1)} \\ \bar{r}_{B,6}^{(2,2)} \end{array} \right), \quad \bar{r}_{C_X} = \left(\begin{array}{c} \bar{r}_{C,5}^{(2,1)} \\ \bar{r}_{C,6}^{(2,2)} \end{array} \right).$$

  Cada par de bloques $(\bar{r}_{B,j}^{(2,i)}, \bar{r}_{C,j}^{(2,i)})$, con $i = 1, 2$, $j = i + 4$, se puede obtener de forma independiente aplicando la matriz de permutación $(P_{jj}^{(2,i)})^T$ (ver (2.98)) sobre $\hat{r}_j^{(2,i)}$

$$(P_{jj}^{(2,i)})^T \hat{r}_j^{(2,i)} \equiv \left(\begin{array}{c} \bar{r}_{B,j}^{(2,i)} \\ \bar{r}_{C,j}^{(2,i)} \end{array} \right). \tag{3.58}$$

  Por otra parte, el bloque $\bar{r}_{C_Z}$ de $\bar{r}$ se obtiene directamente a partir de $\hat{r}_Z$

$$\bar{r}_{C_Z} = \left(\begin{array}{c} \bar{r}_{C,7}^{(1,1)} \end{array} \right) = \left(\begin{array}{c} \hat{r}_7^{(1,1)} \end{array} \right).$$

- En el paso 4, la solución del sistema triangular inferior

$$\left(\begin{array}{cc} \tilde{L}_{B_X} & 0 \\ \tilde{L}_{F_X} & I \end{array} \right) \left(\begin{array}{c} y_{B_X} \\ y_{C_X} \end{array} \right) = \left(\begin{array}{c} \bar{r}_{B_X} \\ \bar{r}_{C_X} \end{array} \right),$$

  se puede particionar conforme a (ver definición de $\tilde{L}_{B_X}$ y $\tilde{L}_{F_X}$ en (2.96))

$$\left(\begin{array}{cccc} \tilde{L}_{B,55}^{(2,1)} & 0 & 0 & 0 \\ 0 & \tilde{L}_{B,66}^{(2,2)} & 0 & 0 \\ \tilde{L}_{F,55}^{(2,1)} & 0 & I & 0 \\ 0 & \tilde{L}_{F,66}^{(2,2)} & 0 & I \end{array} \right) \left(\begin{array}{c} y_{B,5}^{(2,1)} \\ y_{B,6}^{(2,2)} \\ y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \end{array} \right) = \left(\begin{array}{c} \bar{r}_{B,5}^{(2,1)} \\ \bar{r}_{B,6}^{(2,2)} \\ \bar{r}_{C,5}^{(2,1)} \\ \bar{r}_{C,6}^{(2,2)} \end{array} \right),$$

y, por tanto, cada par de bloques $(y_{B,j}^{(2,i)}, \bar{y}_{C,j}^{(2,i)})$, con $i = 1, 2$, $j = i + 4$, se puede obtener de forma independiente resolviendo el siguiente sistema triangular inferior unidad (ver (2.98)):

$$\begin{pmatrix} \tilde{L}_{B,jj}^{(2,i)} & 0 \\ \tilde{L}_{F,jj}^{(2,i)} & I \end{pmatrix} \begin{pmatrix} y_{B,j}^{(2,i)} \\ y_{C,j}^{(2,i)} \end{pmatrix} = \begin{pmatrix} \bar{r}_{B,j}^{(2,i)} \\ \bar{r}_{C,j}^{(2,i)} \end{pmatrix}. \tag{3.59}$$

Por otra parte, la expresión correspondiente al cálculo del bloque y_{C_Z},

$$y_{C_Z} = \bar{r}_{C_Z} - \tilde{L}_{F_Y} \bar{r}_{B_X},$$

se define como (ver definición de $\tilde{L}_{F_Y}$ en (2.96))

$$y_{C,7}^{(1,1)} = \bar{r}_{C,j}^{(1,1)} - \left[\begin{pmatrix} \tilde{L}_{F,75}^{(2,1)} & \tilde{L}_{F,76}^{(2,2)} \end{pmatrix} \begin{pmatrix} \bar{r}_{B,5}^{(2,1)} \\ \bar{r}_{B,6}^{(2,2)} \end{pmatrix} \right],$$

y, por tanto, el substraendo de esta expresión se puede descomponer como

$$y_{C,7}^{(1,1)} = \bar{r}_{C,j}^{(1,1)} - \left[\begin{pmatrix} \tilde{L}_{F,75}^{(2,1)} \bar{r}_{B,5}^{(2,1)} \end{pmatrix} + \begin{pmatrix} \tilde{L}_{F,76}^{(2,2)} \bar{r}_{B,6}^{(2,2)} \end{pmatrix} \right]. \tag{3.60}$$

El cálculo de cualquiera de estos sumandos se puede realizar en paralelo, aunque la acumulación de éstos a y_{C_Z} requiere una sincronización (consultar el Apartado 3.3.2).

Posteriormente, se procede con el siguiente nivel, repitiendo los pasos 1, 2, 3 y 4 con

$$r \equiv y_C = \left(\begin{array}{c} y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \\ \hline\hline y_{C,7}^{(1,1)} \end{array} \right) \equiv \left(\begin{array}{c} r_X \\ \hline\hline r_Z \end{array} \right),$$

hasta que finalmente se alcanza el nivel correspondiente a la última factorización aproximada calculada por el Algoritmo 2.19 en la segunda fase; ver paso 5.

Tras efectuar las computaciones de la segunda fase, el Algoritmo 3.10 procede, en el paso 6, con la tercera y última fase. La ordenación del vector de entrada a la tercera fase se obtiene aplicando Φ^T sobre el vector y_C

$$r \equiv \Phi^T y_C = \Phi^T \left(\begin{array}{c} y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \\ \hline\hline y_{C,7}^{(1,1)} \end{array} \right) = \begin{pmatrix} y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \\ y_{C,7}^{(1,1)} \end{pmatrix} \equiv \left(\begin{array}{c} r_X \\ \hline\hline r_Z \end{array} \right), \tag{3.61}$$

donde el bloque r_X contiene las componentes correspondientes a la última fase (separador del primer nivel), y r_Z es un bloque vacío.

La tercera fase parte del siguiente nivel con r en (3.61) como vector de entrada, y repite los pasos 1, 2, 3, 4 y 5 hasta alcanzar la última factorización aproximada calculada por el Algoritmo 2.19. En esta fase no se puede explotar el tipo de paralelismo descrito para la primera y segunda fase y, además, los bloques que aparecen en el Algoritmo 3.10 con subíndice Z, C_Z o F_Y son vacíos. Si la última factorización aproximada es de la forma (3.37), entonces, en el último nivel, los bloques con subíndice F_X o C_X son vacíos. Por el contrario, si esta factorización es de la forma (2.77), entonces los bloques con subíndice F_X o C_X no son vacíos y, por tanto, una vez calculado y_{C_X}, este bloque se emplea como vector de términos independientes en el paso 7 del Algoritmo 3.11.

Resolución regresiva

La parte regresiva del proceso de aplicación del precondicionador parte del último nivel construido por el Algoritmo 2.19. El vector de salida de este último nivel, $\tilde{y}_C$, se obtiene en el paso 7 del Algoritmo 3.11. En concreto, si y_C no es un bloque vacío, entonces el paso 7 resuelve el sistema de ecuaciones $L_C D_C L_C \tilde{y}_C = y_C$ y, posteriormente, proporciona $\tilde{y}_C$ al nivel anterior reparticionándolo como

$$\tilde{y}_C \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array} \right), \tag{3.62}$$

donde $\tilde{y}_{C_Z}$ es un bloque vacío. Por el contrario, si y_C es un bloque vacío, entonces el paso 7 define $\tilde{y}_C$ como un bloque vacío.

El proceso de la tercera fase se inicia con el nivel anterior repitiendo los pasos 8, 9, 10, 11 y 12 hasta que finalmente se alcanza el primer nivel construido en la tercera fase del Algoritmo 3.11. El vector z que se obtiene en el paso 11 de este primer nivel se puede particionar conforme a $\Phi^T y_C$ en (3.61), obteniendo

$$z = \begin{pmatrix} z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ z_{C,7}^{(1,1)} \end{pmatrix} \equiv \left(\begin{array}{c} z_X \\ \hline z_Z \end{array} \right), \tag{3.63}$$

donde z_Z es un bloque vacío.

Tras efectuar las computaciones de la tercera fase, el Algoritmo 3.11 procede con la fase anterior (segunda fase). La ordenación del vector de salida de la tercera fase, $\tilde{y}_C$, debe coincidir con la ordenación de la segunda fase. Con este propósito, la tercera fase recupera esta ordenación aplicando, en el paso 13, la matriz permutación Φ a z (ver (3.63))

$$\tilde{y}_C \equiv \Phi z = \Phi \begin{pmatrix} z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ z_{C,7}^{(1,1)} \end{pmatrix} = \left(\begin{array}{c} z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ \hline\hline z_{C,7}^{(1,1)} \end{array} \right) \equiv \left(\begin{array}{c} \tilde{y}_{C,5}^{(2,1)} \\ \tilde{y}_{C,6}^{(2,2)} \\ \hline\hline \tilde{y}_{C,7}^{(1,1)} \end{array} \right) \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline\hline \tilde{y}_{C_Z} \end{array} \right). \tag{3.64}$$

La segunda fase procede con el nivel anterior realizando las computaciones de los pasos 8, 9, 10 y 11. La concurrencia que se puede explotar en los pasos 8 y 9 es la siguiente:

- En el paso 8, la operación $\tilde{y}_{B_X} \leftarrow \tilde{D}_{B_X}^{-1} y_{B_X}$ se puede particionar como

$$\begin{pmatrix} \tilde{y}_{B,5}^{(2,1)} \\ \tilde{y}_{B,6}^{(2,2)} \end{pmatrix} \leftarrow \begin{pmatrix} (\tilde{D}_{B,55}^{(2,1)})^{-1} & 0 \\ 0 & (\tilde{D}_{B,66}^{(2,2)})^{-1} \end{pmatrix} \begin{pmatrix} y_{B,5}^{(2,1)} \\ y_{B,6}^{(2,2)} \end{pmatrix},$$

  y, por tanto, cada bloque $\tilde{y}_{B,j}^{(2,i)}$, con $i = 1, 2$, $j = i + 4$, se puede calcular de forma independiente como (ver (2.98))

$$\tilde{y}_{B,j}^{(2,i)} \leftarrow (\tilde{D}_{B,jj}^{(2,i)})^{-1} y_{B,j}^{(2,i)}. \tag{3.65}$$

  Por otra parte, teniendo en cuenta que el bloque $\bar{z}_{C_Z}$ de la solución del sistema (3.50) se calcula *directamente* a partir de $\tilde{y}_{C_Z}$, entonces los bloques $\bar{z}_{B_X}$ y $\bar{z}_{C_X}$ se obtienen resolviendo

$$\begin{pmatrix} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B_X} \\ \bar{z}_{C_X} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B_X} - \tilde{L}_{F_Y}^T \tilde{y}_{C_Z} \\ \tilde{y}_{C_X} \end{pmatrix}.$$

  Este sistema se puede particionar como (ver definición de $\tilde{L}_{B_X}$, $\tilde{L}_{F_X}$ y $\tilde{L}_{F_Y}$ en (2.96))

$$\begin{pmatrix} (\tilde{L}_{B,55}^{(2,1)})^T & 0 & (\tilde{L}_{F,55}^{(2,1)})^T & 0 \\ 0 & (\tilde{L}_{B,66}^{(2,2)})^T & 0 & (\tilde{L}_{F,66}^{(2,2)})^T \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B,5}^{(2,1)} \\ \bar{z}_{B,6}^{(2,2)} \\ \bar{z}_{C,5}^{(2,1)} \\ \bar{z}_{C,6}^{(2,2)} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B,5}^{(2,1)} - (\tilde{L}_{F,75}^{(2,1)})^T \tilde{y}_{C,7}^{(1,1)} \\ \tilde{y}_{B,6}^{(2,2)} - (\tilde{L}_{F,76}^{(2,2)})^T \tilde{y}_{C,7}^{(1,1)} \\ \tilde{y}_{C,5}^{(2,1)} \\ \tilde{y}_{C,6}^{(2,2)} \end{pmatrix},$$

y, por tanto, cada par de bloques $(\bar{z}_{B,j}^{(2,i)}, \bar{z}_{C,j}^{(2,i)})$, con $i = 1, 2$, $j = i + 4$, se puede obtener de forma independiente resolviendo el siguiente sistema triangular superior unidad (ver (2.98)):

$$\begin{pmatrix} (\tilde{L}_{B,jj}^{(2,i)})^T & (\tilde{L}_{F,jj}^{(2,i)})^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B,j}^{(2,i)} \\ \bar{z}_{C,j}^{(2,i)} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B,j}^{(2,i)} - (\tilde{L}_{F,7j}^{(2,i)})^T \tilde{y}_{C,7}^{(1,1)} \\ \tilde{y}_{C,j}^{(2,i)} \end{pmatrix}. \tag{3.66}$$

- El paso 9 aplica la matriz de permutación $\mathrm{diag}\left(\begin{array}{c|c} P_X & I \end{array} \right)$ al vector $\bar{z}$ y, posteriormente, reparticiona el resultado, $\hat{z}$, conforme a la matriz

$$\left(\begin{array}{c||c} \hat{A}_X & \hat{A}_Y^T \\ \hline\hline \hat{A}_Y & A_Z \end{array} \right)$$

en (3.50), obteniendo

$$\hat{z} = \begin{pmatrix} \hat{z}_5^{(2,1)} \\ \hat{z}_6^{(2,2)} \\ \hline\hline \hat{z}_7^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} \hat{z}_X \\ \hline\hline \hat{z}_Z \end{pmatrix}.$$

El bloque $\hat{z}_7^{(1,1)}$ de este vector se obtiene *directamente* a partir de $\bar{z}_{C,7}^{(1,1)}$, y cada bloque $\hat{z}_j^{(2,i)}$, con $i = 1, 2$, $j = i + 4$, se puede obtener de forma independiente aplicando la matriz de permutación $P_{jj}^{(2,i)}$ (ver (2.98)) sobre el par de bloques $(\bar{z}_{B,j}^{(2,i)}, \bar{z}_{C,j}^{(2,i)})$ correspondiente

$$P_{jj}^{(2,i)} \begin{pmatrix} \bar{z}_{B,j}^{(2,i)} \\ \bar{z}_{C,j}^{(2,i)} \end{pmatrix} \equiv \hat{z}_j^{(2,i)}.$$

Posteriormente, se procede con el nivel anterior repitiendo los pasos 8, 9, 10, 11 y 12 con

$$\tilde{y}_C \equiv z = \begin{pmatrix} z_5^{(2,1)} \\ z_6^{(2,2)} \\ \hline\hline z_7^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} \tilde{y}_{C_X} \\ \hline\hline \tilde{y}_{C_Z} \end{pmatrix},$$

hasta alcanzar finalmente el primer nivel construido en la segunda fase del Algoritmo 3.11. El vector z que se obtiene en este primer nivel se puede particionar conforme a $\Phi^T y_C$ en (3.57), obteniendo

$$z = \begin{pmatrix} z_{C,1}^{(3,1)} \\ z_{C,2}^{(3,2)} \\ z_{C,5}^{(2,1)} \\ z_{C,3}^{(3,3)} \\ z_{C,4}^{(3,4)} \\ z_{C,6}^{(2,2)} \\ \hline\hline z_{C,7}^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} z_X \\ \hline\hline z_Z \end{pmatrix}. \tag{3.67}$$

Tras efectuar las computaciones de la segunda fase, el Algoritmo 3.10 recupera la ordenación

de la primera fase aplicando la matriz de permutación Φ sobre el vector z en (3.67) (ver paso 13)

$$\tilde{y}_C \equiv \Phi z = \Phi \begin{pmatrix} z_{C,1}^{(3,1)} \\ z_{C,2}^{(3,2)} \\ z_{C,5}^{(2,1)} \\ z_{C,3}^{(3,3)} \\ z_{C,4}^{(3,4)} \\ z_{C,6}^{(2,2)} \\ \hline\hline z_{C,7}^{(1,1)} \end{pmatrix} = \begin{pmatrix} z_{C,1}^{(3,1)} \\ z_{C,2}^{(3,2)} \\ z_{C,3}^{(3,3)} \\ z_{C,4}^{(3,4)} \\ \hline z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ \hline\hline z_{C,7}^{(1,1)} \end{pmatrix} \equiv \begin{pmatrix} \tilde{y}_{C_X} \\ \hline\hline \tilde{y}_{C_Z} \end{pmatrix}.$$

La primera fase procede con el nivel anterior, realizando los cálculos de los pasos 8, 9, 10 y 11. La concurrencia que se puede explotar en los pasos 8 y 9 se detalla a continuación:

- En el paso 8, la operación $\tilde{y}_{B_X} \leftarrow \tilde{D}_{B_X}^{-1} y_{B_X}$ se puede particionar conforme a

$$\begin{pmatrix} \tilde{y}_{B,1}^{(3,1)} \\ \tilde{y}_{B,2}^{(3,2)} \\ \tilde{y}_{B,3}^{(3,3)} \\ \tilde{y}_{B,4}^{(3,4)} \end{pmatrix} \leftarrow \begin{pmatrix} (\tilde{D}_{B,11}^{(3,1)})^{-1} & 0 & 0 & 0 \\ 0 & (\tilde{D}_{B,22}^{(3,2)})^{-1} & 0 & 0 \\ 0 & 0 & (\tilde{D}_{B,33}^{(3,3)})^{-1} & 0 \\ 0 & 0 & 0 & (\tilde{D}_{B,44}^{(3,4)})^{-1} \end{pmatrix} \begin{pmatrix} y_{B,1}^{(3,1)} \\ y_{B,2}^{(3,2)} \\ y_{B,3}^{(3,3)} \\ y_{B,4}^{(3,4)} \end{pmatrix},$$

  y, por tanto, cada bloque $\tilde{y}_{B,i}^{(3,i)}$, con $i = 1, \ldots, 4$, se puede obtener de forma independiente como (ver (2.88))

$$\tilde{y}_{B,i}^{(3,i)} \leftarrow (\tilde{D}_{B,ii}^{(3,i)})^{-1} y_{B,i}^{(3,i)}. \tag{3.68}$$

  Por otra parte, el bloque $\bar{z}_{C_Z}$ de la solución del sistema (3.50) se obtiene directamente a partir de $\tilde{y}_{C_Z}$

$$\bar{z}_{C_Z} = \begin{pmatrix} \bar{z}_{C,5}^{(2,1)} \\ \bar{z}_{C,6}^{(2,2)} \\ \hline\hline \bar{z}_{C,7}^{(1,1)} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{C,5}^{(2,1)} \\ \tilde{y}_{C,6}^{(2,2)} \\ \hline\hline \tilde{y}_{C,7}^{(1,1)} \end{pmatrix}. \tag{3.69}$$

  Teniendo en cuenta que en el sistema de ecuaciones

$$\begin{pmatrix} \tilde{L}_{B_X}^T & \tilde{L}_{F_X}^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B_X} \\ \bar{z}_{C_X} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B_X} - \tilde{L}_{F_Y}^T \tilde{y}_{C_Z} \\ \tilde{y}_{C_X} \end{pmatrix},$$

  los bloques $\tilde{L}_{B_X}^T$, $\tilde{L}_{F_X}^T$ y $\tilde{L}_{F_Y}^T$ se definen en (2.89) y (2.91), y los bloques $\bar{z}_{B_X}$ y $\bar{z}_{C_X}$ como

$$\bar{z}_{B_X} = \begin{pmatrix} \bar{z}_{B,1}^{(3,1)} \\ \bar{z}_{B,2}^{(3,2)} \\ \bar{z}_{B,3}^{(3,3)} \\ \bar{z}_{B,4}^{(3,4)} \end{pmatrix}, \quad \bar{z}_{C_X} = \begin{pmatrix} \bar{z}_{C,1}^{(3,1)} \\ \bar{z}_{C,2}^{(3,2)} \\ \bar{z}_{C,3}^{(3,3)} \\ \bar{z}_{C,4}^{(3,4)} \end{pmatrix},$$

  entonces cada par de bloques $(\bar{z}_{B,i}^{(3,i)}, \bar{z}_{C,i}^{(3,i)})$, con $i = 1, \ldots, 4$, se puede obtener de forma independiente resolviendo los siguientes sistemas triangulares superiores unidad:

$$\begin{pmatrix} (\tilde{L}_{B,ii}^{(3,i)})^T & (\tilde{L}_{F,ii}^{(3,i)})^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B,i}^{(3,i)} \\ \bar{z}_{C,i}^{(3,i)} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B,i}^{(3,i)} - (\tilde{L}_{F,5i}^{(3,i)})^T \tilde{y}_{C,5}^{(2,i)} - (\tilde{L}_{F,7i}^{(3,i)})^T \tilde{y}_{C,7}^{(1,1)} \\ \tilde{y}_{C,i}^{(3,i)} \end{pmatrix},$$

  para $i = 1, 2$, y

$$\begin{pmatrix} (\tilde{L}_{B,ii}^{(3,i)})^T & (\tilde{L}_{F,ii}^{(3,i)})^T \\ 0 & I \end{pmatrix} \begin{pmatrix} \bar{z}_{B,i}^{(3,i)} \\ \bar{z}_{C,i}^{(3,i)} \end{pmatrix} = \begin{pmatrix} \tilde{y}_{B,i}^{(3,i)} - (\tilde{L}_{F,6i}^{(3,i)})^T \tilde{y}_{C,6}^{(2,i)} - (\tilde{L}_{F,7i}^{(3,i)})^T \tilde{y}_{C,7}^{(1,1)} \\ \tilde{y}_{C,i}^{(3,i)} \end{pmatrix},$$

  para $i = 3, 4$.

- En el paso 9, el vector $\hat{z}$ se reparticiona conforme a la matriz

$$\left(\begin{array}{c|c} \hat{A}_X & \hat{A}_Y^T \\ \hline \hat{A}_Y & A_Z \end{array}\right)$$

en (2.93), obteniéndose

$$\hat{z} = \left(\begin{array}{c} \hat{z}_1^{(3,1)} \\ \hat{z}_2^{(3,2)} \\ \hat{z}_3^{(3,3)} \\ \hat{z}_4^{(3,4)} \\ \hline \hat{z}_5^{(2,1)} \\ \hat{z}_6^{(2,2)} \\ \hline\hline \hat{z}_7^{(1,1)} \end{array}\right) \equiv \left(\begin{array}{c} \hat{z}_X \\ \hline \hat{z}_Z \end{array}\right).$$

El bloque $\hat{z}_Z$ de este vector se calcula directamente a partir de $\bar{z}_{C_Z}$ (ver (3.69)), y cada bloque de $\hat{z}_X$, $\hat{z}_i^{(3,i)}$, con $i = 1, \ldots, 4$, se puede obtener de forma independiente aplicando la matriz de permutación $P_{ii}^{(3,i)}$ (ver (2.88)) sobre el par de bloques $(\bar{z}_{B,i}^{(3,i)}, \bar{z}_{C,i}^{(3,i)})$ correspondiente

$$P_{ii}^{(3,i)} \left(\begin{array}{c} \bar{z}_{B,i}^{(3,i)} \\ \bar{z}_{C,i}^{(3,i)} \end{array}\right) \equiv \hat{z}_j^{(2,i)}.$$

Posteriormente, se procede con el nivel anterior con

$$\tilde{y}_C \equiv z = \left(\begin{array}{c} z_1^{(3,1)} \\ z_2^{(3,2)} \\ z_3^{(3,3)} \\ z_4^{(3,4)} \\ \hline z_5^{(2,1)} \\ z_6^{(2,2)} \\ \hline\hline z_7^{(1,1)} \end{array}\right) \equiv \left(\begin{array}{c} \tilde{y}_{C_X} \\ \hline \tilde{y}_{C_Z} \end{array}\right),$$

repitiendo los pasos 8, 9, 10, 11 y 12 hasta que finalmente el proceso alcanza el primer nivel construido por el Algoritmo 2.19. En este punto, el Algoritmo 3.11 devuelve la solución del sistema $Mz = r$, aplicando la matriz de permutación Ω sobre el vector z obtenido finalmente en el paso 11; ver **Salida** del Algoritmo 3.10.

3.3.2. Aprovechamiento del paralelismo a nivel del árbol de tareas

La aplicación paralela del precondicionador multinivel explota la concurrencia que se pone de manifiesto en los Algoritmos 3.10 y 3.11. Dicha concurrencia se captura a través de los árboles de dependencias entre tareas mostrados en la Figura 3.1 (a) (resolución progresiva) y 3.1 (b) (resolución regresiva). Estos árboles descomponen el proceso de aplicación del precondicionador en tareas (nodos) y determinan el orden en el que éstas deben ejecutarse, identificando las dependencias (arcos) existentes entre las mismas. Teniendo en cuenta el sentido de los arcos en los árboles de la Figura 3.1, la resolución progresiva paralela se inicia con la ejecución de las tareas ligadas a las hojas del árbol y procede resolviendo progresivamente las dependencias de las tareas hasta llegar a la raíz; a continuación, la resolución regresiva paralela comienza con la ejecución de la tarea raíz y procede en sentido inverso, hasta llegar a las hojas del árbol. En lo que se refiere a la relación de parentesco entre tareas, en este apartado mantendremos la nomenclatura del Apartado 2.3.3 (por ejemplo, $(3,1)$ y $(3,2)$ son los hijos de $(2,1)$, y $(2,1)$ es el padre de $(3,1)$ y $(3,2)$, independientemente del sentido de los arcos).

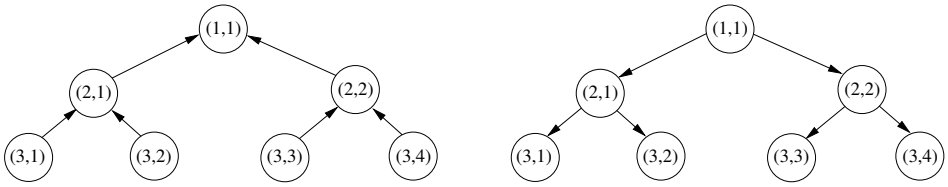


Figura 3.1: Árboles de dependencias entre tareas para el proceso paralelo de aplicación del precondicionador. (a) Resolución progresiva. (b) Resolución regresiva.

Resolución progresiva paralela

La resolución progresiva paralela se inicia con las tareas hoja del árbol. Estas tareas realizan, de modo independiente, los cálculos de la primera fase del Algoritmo 3.10, explotando para ello el paralelismo disponible en los pasos 1, 2, 3 y 4. En el paso 3, cada tarea hoja, $(3,i)$, para $i = 1,\ldots,4$, obtiene el par de bloques $(\bar{r}_{B,i}^{(3,i)}, \bar{r}_{C,i}^{(3,i)})$ de $\bar{r}_{B_X}$ y $\bar{r}_{C_X}$ conforme a (3.53) y, posteriormente, en el paso 4, el par de bloques $(y_{B,i}^{(3,i)}, y_{C,i}^{(3,i)})$ de y_{B_X} e y_{C_X} se obtiene resolviendo el sistema triangular inferior (3.55). El cálculo paralelo del bloque y_{C_Z} (ver (3.56)) se gestiona teniendo en cuenta las mismas consideraciones que se aplicaron para la construcción de $\hat{S}_{C_Z}$; ver Apartado 2.3.3. Por lo tanto, es necesario descomponer el vector residuo de entrada para la primera fase, $r \equiv \Omega^T r$, en cuatro sumandos

$$r \equiv \Omega^T r = \begin{pmatrix} r_1^{(3,1)} \\ r_2^{(3,2)} \\ r_3^{(3,3)} \\ r_4^{(3,4)} \\ \hline r_5^{(2,1)} \\ r_6^{(2,2)} \\ \hline\hline r_7^{(1,1)} \end{pmatrix} = \begin{pmatrix} r_1^{(3,1)} \\ 0 \\ 0 \\ 0 \\ \hline r_5^{(3,1)} \\ 0 \\ \hline\hline r_7^{(3,1)} \end{pmatrix} + \begin{pmatrix} 0 \\ r_2^{(3,2)} \\ 0 \\ 0 \\ \hline r_5^{(3,2)} \\ 0 \\ \hline\hline r_7^{(3,2)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ r_3^{(3,3)} \\ 0 \\ \hline 0 \\ r_6^{(3,3)} \\ \hline\hline r_7^{(3,3)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ r_4^{(3,4)} \\ \hline 0 \\ r_6^{(3,4)} \\ \hline\hline r_7^{(3,4)} \end{pmatrix}$$
$$= r^{(3,1)} + r^{(3,2)} + r^{(3,3)} + r^{(3,4)},$$

donde cada sumando está compuesto por el bloque correspondiente a la tarea hoja, $r_i^{(3,i)}$, junto con los bloques locales correspondientes a sus ancestros. Por ejemplo, en el caso de $r^{(3,2)}$, $r_5^{(3,2)}$ y $r_7^{(3,2)}$ son los bloques locales de $(3,2)$ correspondientes a sus respectivos ancestros $(2,1)$ y $(1,1)$.

Para realizar en paralelo las computaciones de la primera fase del Algoritmo 3.10, cada tarea hoja $(3,i)$ procesa $r^{(3,i)}$ aplicando *localmente* el siguiente algoritmo:

- Los pasos 1 y 2 aplican la matriz de escalado diagonal, $D^{(3,i)}$, $r^{(3,i)} \rightarrow D^{(3,i)} r^{(3,i)} = \tilde{r}^{(3,i)}$, y la matriz de permutación, $(\tilde{P}^{(3,i)})^T$, $\tilde{r}^{(3,i)} \rightarrow (\tilde{P}^{(3,i)})^T \tilde{r}^{(3,i)} = \hat{r}^{(3,i)}$. Particionando estas expresiones, obtenemos (para el caso de la tarea $(3,2)$)

$$\begin{pmatrix} \tilde{r}_2^{(3,2)} \\ \hline \tilde{r}_5^{(3,2)} \\ \tilde{r}_7^{(3,2)} \end{pmatrix} = \left(\begin{array}{c|cc} D_{22}^{(3,2)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \begin{pmatrix} r_2^{(3,2)} \\ \hline r_5^{(3,2)} \\ r_7^{(3,2)} \end{pmatrix}, \quad \begin{pmatrix} \hat{r}_2^{(3,2)} \\ \hline \hat{r}_5^{(3,2)} \\ \hat{r}_7^{(3,2)} \end{pmatrix} = \left(\begin{array}{c|cc} (\tilde{P}_{22}^{(3,2)})^T & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \begin{pmatrix} \tilde{r}_2^{(3,2)} \\ \hline \tilde{r}_5^{(3,2)} \\ \tilde{r}_7^{(3,2)} \end{pmatrix},$$

  y, por tanto, los bloques de $\tilde{r}^{(3,2)}$ y $\hat{r}^{(3,2)}$ correspondientes a los ancestros de $(3,2)$ se obtienen directamente a partir de los bloques de $r^{(3,2)}$ (por ejemplo, $\tilde{r}_7^{(3,2)}$ y $\hat{r}_7^{(3,2)}$ se obtienen directamente a partir de $r_7^{(3,2)}$).

- El paso 3 aplica la matriz de permutación $(P^{(3,i)})^T$ al vector $\hat{r}^{(3,i)}$, reparticionando el resultado conforme a la factorización aproximada por bloques (2.113). Esta operación se

puede particionar para el caso de la tarea $(3,2)$ como sigue

$$\left(\begin{array}{c|cc} (P_{22}^{(3,2)})^T & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right)\left(\begin{array}{c} \hat{r}_2^{(3,2)} \\ \hline \hat{r}_5^{(3,2)} \\ \hat{r}_7^{(3,2)} \end{array}\right) \equiv \left(\begin{array}{c} \bar{r}_{B,2}^{(3,2)} \\ \bar{r}_{C,2}^{(3,2)} \\ \hline \bar{r}_{C,5}^{(3,2)} \\ \bar{r}_{C,7}^{(3,2)} \end{array}\right).$$

De esta expresión se deduce que los bloques de $\bar{r}^{(3,2)}$ correspondientes a los ancestros de $(3,2)$ se obtienen directamente a partir de los bloques de $\hat{r}^{(3,2)}$.

- El paso 4 resuelve el sistema triangular inferior (para el caso de la tarea $(3,2)$)

$$\left(\begin{array}{cc|cc} \tilde{L}_{B,22}^{(3,2)} & 0 & 0 & 0 \\ \tilde{L}_{F,22}^{(3,2)} & I & 0 & 0 \\ \hline \tilde{L}_{F,52}^{(3,2)} & 0 & I & 0 \\ \tilde{L}_{F,72}^{(3,2)} & 0 & 0 & I \end{array}\right)\left(\begin{array}{c} y_{B,2}^{(3,2)} \\ y_{C,2}^{(3,2)} \\ \hline y_{C,5}^{(3,2)} \\ y_{C,7}^{(3,2)} \end{array}\right) = \left(\begin{array}{c} \bar{r}_{B,2}^{(3,2)} \\ \bar{r}_{C,2}^{(3,2)} \\ \hline \bar{r}_{C,5}^{(3,2)} \\ \bar{r}_{C,7}^{(3,2)} \end{array}\right), \tag{3.70}$$

donde $y_{C,5}^{(3,2)}$ e $y_{C,7}^{(3,2)}$ son las contribuciones locales de la tarea $(3,2)$ a los bloques $y_{C,5}^{(2,1)}$ e $y_{C,7}^{(1,1)}$ de y_{C_Z}; ver (3.56). Las entradas originales de los bloques de y_{C_Z} se pueden recuperar sumando los bloques locales de contribución de las tareas hoja. Por ejemplo, el bloque $y_{C,5}^{(2,1)}$ se puede recuperar sumando los bloques locales de contribución de $(3,1)$ y $(3,2)$ conforme a

$$\begin{aligned} y_{C,5}^{(2,1)} &= y_{C,5}^{(3,1)} + y_{C,5}^{(3,2)} \\ &= \bar{r}_{C,5}^{(3,1)} - \tilde{L}_{F,51}^{(3,1)} y_{B,1}^{(3,1)} + \bar{r}_{C,5}^{(3,2)} - \tilde{L}_{F,52}^{(3,2)} y_{B,2}^{(3,2)} \\ &= \bar{r}_{C,5}^{(2,1)} - \tilde{L}_{F,51}^{(3,1)} y_{B,1}^{(3,1)} - \tilde{L}_{F,51}^{(3,1)} y_{B,1}^{(3,1)}, \end{aligned} \tag{3.71}$$

y el bloque $y_{C,7}^{(1,1)}$ sumando las contribuciones locales de todas las tareas hoja como

$$y_{C,7}^{(1,1)} = y_{C,7}^{(3,1)} + y_{C,7}^{(3,2)} + y_{C,7}^{(3,3)} + y_{C,7}^{(3,4)}. \tag{3.72}$$

Como veremos más adelante, el algoritmo paralelo sólo suma los bloques locales de contribución de las tareas en el momento en el que el proceso de resolución progresiva necesita recuperar la entradas de los bloques de y_{C_Z}.

- El paso 5 procede con el siguiente nivel repitiendo los pasos 1, 2, 3 y 4 con $r^{(3,i)} \equiv y_C^{(3,i)}$, hasta que finalmente (3.70) se corresponde con la última factorización aproximada por bloques construida localmente por la tarea. Por último, cada tarea hoja "envía" el vector $y_C^{(3,i)}$ a su padre.

Tras los cálculos correspondientes a las hojas, la resolución progresiva paralela procede con las tareas del segundo nivel. Estas tareas realizan, de modo independiente, los cálculos de la segunda fase del Algoritmo 3.10. En el paso 3, cada tarea hoja, $(2,i)$, para $i = 1,2$, obtiene el par de bloques $(\bar{r}_{B,j}^{(2,i)}, \bar{r}_{C,j}^{(2,i)})$, con $j = i+4$, conforme a (3.58) y, posteriormente, en el paso 4, el par de bloques $(y_{B,j}^{(2,i)}, y_{C,j}^{(2,i)})$ se obtiene resolviendo el sistema triangular inferior (3.59). En este paso, las tareas también calculan sus contribuciones locales a los bloques de y_{C_Z} (ver (3.60)) y, por tanto, cada tarea del segundo nivel debe obtener un sumando de (3.57) que contenga, además del bloque $r_j^{(2,i)}$, el bloque local correspondiente a su ancestro $(1,1)$. Estos sumandos se

definen como

$$\begin{array}{rcl} r & = & r^{(2,1)} + r^{(2,2)} \\ \left(\begin{array}{c} r_5^{(2,1)} \\ \hline\hline r_6^{(2,2)} \\ \hline\hline r_7^{(1,1)} \end{array}\right) & = & \left(\begin{array}{c} r_5^{(2,1)} \\ \hline\hline 0 \\ \hline\hline r_7^{(2,1)} \end{array}\right) + \left(\begin{array}{c} 0 \\ \hline\hline r_6^{(2,2)} \\ \hline\hline r_7^{(2,2)} \end{array}\right). \end{array}$$

Si reparticionamos los bloques $r_5^{(2,1)}$ y $r_6^{(2,2)}$ conforme a la definición de r en (3.57)

$$r_5^{(2,1)} = \left(\begin{array}{c} y_{C,1}^{(3,1)} \\ y_{C,2}^{(3,2)} \\ \mathbf{y_{C,5}^{(2,1)}} \end{array}\right), \; r_6^{(2,2)} = \left(\begin{array}{c} y_{C,3}^{(3,3)} \\ y_{C,4}^{(3,4)} \\ \mathbf{y_{C,6}^{(2,2)}} \end{array}\right),$$

podemos observar, por una parte, que las tareas del segundo nivel pueden obtener los bloques que no están marcados en negrita *directamente* a partir de los vectores calculados por sus hijos en el paso 4 (ver (3.70)) y, por otra, que las entradas de los bloques marcados en negrita se recuperan sumando sus bloques locales de contribución; es decir, $(2,1)$ obtiene $y_{C,5}^{(2,1)}$ conforme a (3.71), y $(2,2)$ hace lo propio con $y_{C,6}^{(2,2)}$, calculando $y_{C,6}^{(2,2)} = y_{C,6}^{(3,3)} + y_{C,6}^{(3,4)}$. Por último, teniendo en cuenta que $r_7^{(1,1)} = y_{C,7}^{(1,1)}$ (ver (3.57)) y (3.72), entonces $(2,1)$ y $(2,2)$ obtienen el bloque local correspondiente a su ancestro $(1,1)$ sumando los bloques de contribución de sus hijos; es decir, la tarea $(2,1)$ obtiene $r_7^{(2,1)}$ conforme a

$$r_7^{(2,1)} = y_{C,7}^{(3,1)} + y_{C,7}^{(3,2)},$$

y $(2,2)$ obtiene $r_7^{(2,2)}$ como

$$r_7^{(2,2)} = y_{C,7}^{(3,3)} + y_{C,7}^{(3,4)}.$$

Tras construir $r^{(2,i)}$, las tareas $(2,i)$, con $i = 1,2$, realizan de modo independiente las computaciones de la segunda fase del Algoritmo 3.10 aplicando *localmente* el siguiente algoritmo:

- Los pasos 1 y 2 aplican la matriz de escalado diagonal, $D^{(2,i)}$, $r^{(2,i)} \to D^{(2,i)} r^{(2,i)} = \tilde{r}^{(2,i)}$, y la matriz de permutación, $(\tilde{P}^{(2,i)})^T$, $\tilde{r}^{(2,i)} \to (\tilde{P}^{(2,i)})^T \tilde{r}^{(2,i)} = \hat{r}^{(2,i)}$. Particionando estas expresiones, obtenemos

$$\left(\begin{array}{c} \tilde{r}_j^{(2,i)} \\ \hline \tilde{r}_7^{(2,i)} \end{array}\right) = \left(\begin{array}{c|c} D_{jj}^{(2,i)} & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} r_j^{(2,i)} \\ \hline r_7^{(2,i)} \end{array}\right), \quad \left(\begin{array}{c} \hat{r}_j^{(2,i)} \\ \hline \hat{r}_7^{(2,i)} \end{array}\right) = \left(\begin{array}{c|c} (\tilde{P}_{22}^{(3,2)})^T & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \tilde{r}_j^{(2,i)} \\ \hline \tilde{r}_7^{(2,i)} \end{array}\right).$$

- El paso 3 aplica la matriz de permutación $(P^{(2,i)})^T$ al vector $\hat{r}^{(2,i)}$, reparticionando el resultado conforme a la factorización aproximada por bloques (2.115). Esta operación se puede particionar como

$$\left(\begin{array}{c|c} (P_{jj}^{(2,i)})^T & 0 \\ \hline 0 & I \end{array}\right) \left(\begin{array}{c} \hat{r}_j^{(2,i)} \\ \hline \hat{r}_7^{(2,i)} \end{array}\right) \equiv \left(\begin{array}{c} \bar{r}_{B,j}^{(2,i)} \\ \bar{r}_{C,j}^{(2,i)} \\ \hline \bar{r}_{C,7}^{(2,i)} \end{array}\right).$$

- El paso 4 resuelve el siguiente sistema triangular inferior unidad

$$\left(\begin{array}{cc|c} \tilde{L}_{B,jj}^{(2,i)} & 0 & 0 \\ \tilde{L}_{F,jj}^{(2,i)} & I & 0 \\ \hline \tilde{L}_{F,7j}^{(2,i)} & 0 & I \end{array}\right) \left(\begin{array}{c} y_{B,j}^{(2,i)} \\ y_{C,j}^{(2,i)} \\ \hline y_{C,7}^{(2,i)} \end{array}\right) = \left(\begin{array}{c} \bar{r}_{B,j}^{(2,i)} \\ \bar{r}_{C,j}^{(2,i)} \\ \hline \bar{r}_{C,7}^{(2,i)} \end{array}\right), \tag{3.73}$$

  donde $y_{C,7}^{(2,i)}$ contiene las contribuciones locales de la tarea $(2,i)$ a y_{C_Z}; ver (3.60).

- El paso 5 procede con el siguiente nivel repitiendo los pasos 1, 2, 3 y 4 con $r^{(2,i)} \equiv y_C^{(2,i)}$, hasta que finalmente (3.73) se corresponde con la última factorización aproximada por bloques construida localmente por la tarea. Por último, cada tarea "envía" el vector $y_C^{(2,i)}$ a su padre.

La tarea raíz $(1,1)$ se encarga de realizar las computaciones de la tercera y última fase del Algoritmo 3.10 y, por tanto, procede desde el primer nivel hasta el último construido localmente, repitiendo los pasos 1, 2, 3, 4 y 5. Para poder realizar estas computaciones, el vector residuo de entrada a la tarea, $r^{(1,1)}$, debe coincidir con el vector residuo de entrada, r, a la tercera fase (ver (3.61))

$$r^{(1,1)} = \begin{pmatrix} y_{C,5}^{(2,1)} \\ y_{C,6}^{(2,2)} \\ \mathbf{y_{C,7}^{(1,1)}} \end{pmatrix}. \tag{3.74}$$

La tarea raíz obtiene *directamente* los bloques $y_{C,5}^{(2,1)}$ e $y_{C,6}^{(2,2)}$ de este vector a partir de los vectores $y_C^{(2,i)}$ calculados por sus hijos en el paso 4 (ver (3.73)) y, el bloque marcado en negrita, sumando sus bloques locales de contribución

$$y_{C,7}^{(1,1)} = y_{C,7}^{(2,1)} + y_{C,7}^{(2,2)}.$$

Resolución regresiva paralela

La regresión regresiva paralela se inicia con la ejecución de la tarea raíz. Esta tarea se encarga de realizar las computaciones de la tercera fase del Algoritmo 3.11 y, por lo tanto, procede desde el último nivel hasta el primero construido localmente, repitiendo los pasos 8, 9, 10, 11 y 12. El vector $z^{(1,1)}$ que se obtiene en el paso 11 de este primer nivel se puede particionar conforme al vector de entrada, $r^{(1,1)}$, obteniéndose (ver (3.74))

$$z^{(1,1)} = \begin{pmatrix} z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ z_{C,7}^{(1,1)} \end{pmatrix},$$

y, la salida de la tarea raíz, $\tilde{y}_C^{(1,1)}$, se obtiene directamente a partir de este vector (ver (3.64))

$$\tilde{y}_C^{(1,1)} = \left(\begin{array}{c} \tilde{y}_{C,5}^{(2,1)} \\ \tilde{y}_{C,6}^{(2,2)} \\ \hline \tilde{y}_{C,7}^{(1,1)} \end{array}\right) = \left(\begin{array}{c} z_{C,5}^{(2,1)} \\ z_{C,6}^{(2,2)} \\ z_{C,7}^{(1,1)} \end{array}\right). \tag{3.75}$$

El particionado anterior de $\tilde{y}_C^{(1,1)}$ separa los bloques correspondientes a los hijos de $(1,1)$, $\tilde{y}_{C,5}^{(2,1)}$ y $\tilde{y}_{C,6}^{(2,2)}$, del bloque correspondiente a su ancestro, $\tilde{y}_{C,7}^{(1,1)}$.

Las tareas del segundo nivel realizan, de modo independiente, las computaciones de la segunda fase del Algoritmo 3.11, explotando para ello el paralelismo que se pone de manifiesto en los pasos 8, 9, 10 y 11. En el paso 8, cada tarea $(2,i)$, con $i = 1, 2$, $j = i + 4$, obtiene, de forma independiente al resto, el bloque $\tilde{y}_{B,j}^{(2,i)}$ conforme a (3.65) y, posteriormente, resuelve el siguiente sistema triangular superior unidad:

$$\left(\begin{array}{cc|c} (\tilde{L}_{B,jj}^{(2,i)})^T & (\tilde{L}_{F,jj}^{(2,i)})^T & (\tilde{L}_{F,7j}^{(2,i)})^T \\ 0 & I & 0 \\ \hline 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \bar{z}_{B,j}^{(2,i)} \\ \bar{z}_{C,j}^{(2,i)} \\ \hline \bar{z}_{C,7}^{(1,1)} \end{array}\right) = \left(\begin{array}{c} \tilde{y}_{B,j}^{(2,i)} \\ \mathbf{\tilde{y}_{C,j}^{(2,i)}} \\ \hline \mathbf{\tilde{y}_{C,7}^{(1,1)}} \end{array}\right). \tag{3.76}$$

En esta última expresión, es posible observar que los bloques marcados en negrita han sido calculados localmente por la tarea $(1,1)$ (ver (3.75)). Por tanto, antes de realizar sus respectivas computaciones, las tareas del segundo nivel deben "recibir" estos bloques de la tarea raíz. Con este propósito, la tarea raíz descompone el vector $\tilde{y}_C^{(1,1)}$ de modo que cada tarea $(2,i)$, con $i=1,2$, $j=i+4$, "recibe" el bloque $\tilde{y}_{C,j}^{(2,i)}$, y un duplicado del bloque $\tilde{y}_{C,7}^{(1,1)}$, es decir, un duplicado del bloque correspondiente a su ancestro $(1,1)$.

Tomando en cuenta las consideraciones anteriores, las tareas del segundo nivel realizan en paralelo las computaciones de la segunda fase del Algoritmo 3.11, aplicando localmente el siguiente algoritmo:

- Inicialmente, la tarea $(2,i)$ "recibe" los bloques $\tilde{y}_{C,j}^{(2,i)}$ e $\tilde{y}_{C,7}^{(1,1)}$ de su padre.
- En el paso 8, la tarea $(2,i)$ calcula $\tilde{y}_{B,j}^{(2,i)}$ y, posteriormente, resuelve el sistema triangular inferior unidad (3.76).
- En el paso 9, la tarea $(2,i)$ aplica la matriz de permutación $P^{(2,i)}$ al vector $\bar{z}^{(2,i)}$, repartiendo el resultado, $\hat{z}^{(2,i)}$, conforme a la matriz $\hat{A}^{(2,i)}$ en (2.115)

$$\left(\begin{array}{c|c} P_{jj}^{(2,i)} & 0 \\ \hline 0 & I \end{array}\right)\left(\begin{array}{c} \bar{z}_{B,j}^{(2,i)} \\ \bar{z}_{C,j}^{(2,i)} \\ \hline \bar{z}_{C,7}^{(1,1)} \end{array}\right) \equiv \left(\begin{array}{c} \hat{z}_j^{(2,i)} \\ \hline \hat{z}_7^{(1,1)} \end{array}\right).$$

- En los pasos 10 y 11, la tarea $(2,i)$ aplica la matriz de permutación, $\tilde{P}^{(2,i)}$, $\hat{z}^{(2,i)} \rightarrow \tilde{P}^{(2,i)}\hat{z}^{(2,i)} = \tilde{z}^{(2,i)}$, y la matriz de escalado diagonal, $D^{(2,i)}$, $\tilde{z}^{(2,i)} \rightarrow D^{(2,i)}\tilde{z}^{(2,i)} = z^{(2,i)}$. Particionando estas expresiones, obtenemos

$$\left(\begin{array}{c} \tilde{z}_j^{(2,i)} \\ \hline \tilde{z}_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c|c} \tilde{P}_{jj}^{(2,i)} & 0 \\ \hline 0 & I \end{array}\right)\left(\begin{array}{c} \hat{z}_j^{(2,i)} \\ \hline \hat{z}_7^{(1,1)} \end{array}\right), \quad \left(\begin{array}{c} z_j^{(2,i)} \\ \hline z_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c|c} D_{jj}^{(2,i)} & 0 \\ \hline 0 & I \end{array}\right)\left(\begin{array}{c} \tilde{z}_j^{(2,i)} \\ \hline \tilde{z}_7^{(1,1)} \end{array}\right).$$

- En el paso 12, la tarea $(2,i)$ procede con el nivel anterior repitiendo los pasos 8, 9, 10 y 11 con $\tilde{y}_C^{(2,i)} \equiv z^{(2,i)}$, hasta que finalmente (3.66) se corresponde con la primera factorización aproximada por bloques construida localmente por la tarea.

El vector $z^{(2,i)}$ calculado en el paso 11 del algoritmo anterior se puede particionar conforme al vector residuo de entrada a la tarea, $r^{(2,i)}$, obteniendo

$$z^{(2,1)} = \left(\begin{array}{c} z_5^{(2,1)} \\ \hline z_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c} z_{C,1}^{(3,1)} \\ z_{C,2}^{(3,2)} \\ z_{C,5}^{(2,1)} \\ \hline z_{C,7}^{(1,1)} \end{array}\right), \quad z^{(2,2)} = \left(\begin{array}{c} z_6^{(2,2)} \\ \hline z_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c} z_{C,3}^{(3,3)} \\ z_{C,4}^{(3,4)} \\ z_{C,6}^{(2,2)} \\ \hline z_{C,7}^{(1,1)} \end{array}\right),$$

y, el vector de salida, $\tilde{y}_C^{(2,i)}$, se obtiene directamente a partir de $z^{(2,i)}$

$$\tilde{y}_C^{(2,1)} = \left(\begin{array}{c} \tilde{y}_{C,1}^{(3,1)} \\ \tilde{y}_{C,2}^{(3,2)} \\ \hline \tilde{y}_{C,5}^{(2,1)} \\ \tilde{y}_{C,7}^{(1,1)} \end{array}\right) = \left(\begin{array}{c} z_{C,1}^{(3,1)} \\ z_{C,2}^{(3,2)} \\ z_{C,5}^{(2,1)} \\ \hline z_{C,7}^{(1,1)} \end{array}\right), \quad \tilde{y}_C^{(2,2)} = \left(\begin{array}{c} \tilde{y}_{C,3}^{(3,3)} \\ \tilde{y}_{C,4}^{(3,4)} \\ \hline \tilde{y}_{C,6}^{(2,2)} \\ \tilde{y}_{C,7}^{(1,1)} \end{array}\right) = \left(\begin{array}{c} z_{C,3}^{(3,3)} \\ z_{C,4}^{(3,4)} \\ z_{C,6}^{(2,2)} \\ \hline z_{C,7}^{(1,1)} \end{array}\right). \tag{3.77}$$

La línea de particionado de $\tilde{y}_C^{(2,i)}$ se ha desplazado una posición hacia arriba con respecto a la de $z^{(2,i)}$ para separar los bloques correspondientes a los hijos de $(2,i)$ de los bloques correspondientes a sus ancestros, $(2,i)$ y $(1,1)$.

Las tareas hoja realizan, de modo independiente, las computaciones de la primera fase del Algoritmo 3.11, explotando para ello el paralelismo disponible en los pasos 8, 9, 10 y 11. En el paso 8, cada tarea $(3,i)$, con $i=1,\ldots,4$, obtiene el bloque $\tilde{y}_{B,i}^{(3,i)}$ conforme a (3.68) y, posteriormente, resuelve el siguiente sistema triangular superior unidad (para el caso de la tarea $(3,2)$)

$$\left(\begin{array}{cc|cc} (\tilde{L}_{B,22}^{(3,2)})^T & (\tilde{L}_{F,22}^{(3,2)})^T & (\tilde{L}_{F,5j}^{(3,2)})^T & (\tilde{L}_{F,5j}^{(3,2)})^T \\ 0 & I & 0 & 0 \\ \hline 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \bar{z}_{B,2}^{(3,2)} \\ \bar{z}_{C,2}^{(3,2)} \\ \hline \bar{z}_{C,5}^{(2,1)} \\ \bar{z}_{C,7}^{(1,1)} \end{array}\right) = \left(\begin{array}{c} \tilde{y}_{B,2}^{(3,2)} \\ \tilde{\mathbf{y}}_{\mathbf{C,2}}^{\mathbf{(3,2)}} \\ \hline \tilde{\mathbf{y}}_{\mathbf{C,5}}^{\mathbf{(2,1)}} \\ \tilde{\mathbf{y}}_{\mathbf{C,7}}^{\mathbf{(1,1)}} \end{array}\right). \tag{3.78}$$

Para poder efectuar este último cálculo, cada tarea hoja debe "recibir" de su padre los bloques marcados en negrita. Con este propósito, cada tarea del segundo nivel, $(2,i)$, con $i=1,2$, descompone su vector de salida, $\tilde{y}_C^{(2,i)}$ (ver (3.77)), de forma que sus hijos, $(3,2i-1)$ y $(3,2i)$, "reciben" los bloques $\tilde{y}_{C,2i-1}^{(3,2i-1)}$ e $\tilde{y}_{C,2i}^{(3,2i)}$, respectivamente, y un duplicado de los bloques correspondientes a sus ancestros; ver Figura 3.2.

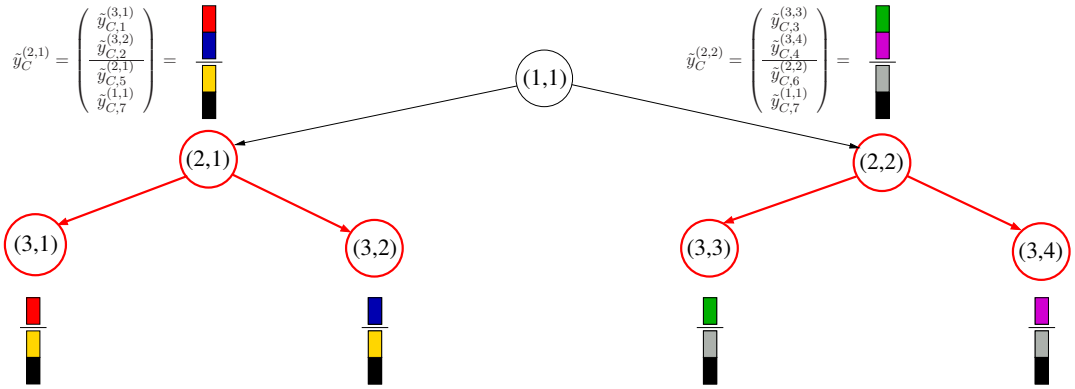


Figura 3.2: Cada tarea del segundo nivel, $(2,i)$, con $i=1,2$, descompone su vector de salida, $\tilde{y}_C^{(2,i)}$, de forma que sus hijos, $(3,2i-1)$ y $(3,2i)$, "reciben" los bloques $\tilde{y}_{C,2i-1}^{(3,2i-1)}$ e $y_{C,2i}^{(3,2i)}$, respectivamente, y un duplicado de los bloques correspondientes a sus ancestros.

Teniendo en cuenta las consideraciones del párrafo anterior, las tareas $(3,i)$, con $i=1,\ldots,4$, realizan de modo independiente las computaciones de la primera fase del Algoritmo 3.11, aplicando localmente el siguiente algoritmo:

- Inicialmente, la tarea $(3,i)$ "recibe" de su padre el bloque $\tilde{y}_{C,i}^{(3,i)}$ y un duplicado de los bloques correspondientes a sus ancestros; ver Figura 3.2.

- En el paso 8, la tarea $(3,i)$ calcula $\tilde{y}_{B,i}^{(3,i)}$ y, posteriormente, resuelve el sistema triangular inferior unidad (3.78).

- En el paso 9, la tarea $(3,i)$ aplica la matriz de permutación $P^{(3,i)}$ al vector $\bar{z}^{(3,i)}$, repartiendo el resultado, $\hat{z}^{(3,i)}$, conforme a la matriz $\hat{A}^{(3,i)}$. En el caso de la tarea $(3,2)$, esta operación se puede particionar como (ver (2.113))

$$\left(\begin{array}{c|cc} P_{22}^{(3,2)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \bar{z}_{B,2}^{(3,2)} \\ \bar{z}_{C,2}^{(3,2)} \\ \hline \bar{z}_{C,5}^{(2,1)} \\ \bar{z}_{C,7}^{(1,1)} \end{array}\right) \equiv \left(\begin{array}{c} \hat{z}_2^{(3,2)} \\ \hline \hat{z}_5^{(2,1)} \\ \hat{z}_7^{(1,1)} \end{array}\right).$$

- En los pasos 10 y 11, la tarea $(3,i)$ aplica la matriz de permutación, $\tilde{P}^{(3,i)}$, $\hat{z}^{(3,i)} \rightarrow \tilde{P}^{(3,i)}\hat{z}^{(3,i)} = \tilde{z}^{(3,i)}$, y la matriz de escalado diagonal, $D^{(3,i)}$, $\tilde{z}^{(3,i)} \rightarrow D^{(3,i)}\tilde{z}^{(3,i)} = z^{(3,i)}$. Particionando estas expresiones, obtenemos (en el caso de la tarea $(3,2)$)

$$\left(\begin{array}{c} \tilde{z}_2^{(3,2)} \\ \hline \tilde{z}_5^{(2,1)} \\ \tilde{z}_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c|cc} \tilde{P}_{22}^{(3,2)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \hat{z}_2^{(3,2)} \\ \hline \hat{z}_5^{(2,1)} \\ \hat{z}_7^{(1,1)} \end{array}\right), \quad \left(\begin{array}{c} z_2^{(3,2)} \\ \hline z_5^{(2,1)} \\ z_7^{(1,1)} \end{array}\right) = \left(\begin{array}{c|cc} D_{22}^{(3,2)} & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right) \left(\begin{array}{c} \tilde{z}_2^{(3,2)} \\ \hline \tilde{z}_5^{(2,1)} \\ \tilde{z}_7^{(1,1)} \end{array}\right),$$

- En el paso 12, la tarea $(3,i)$ procede con el nivel anterior repitiendo los pasos 8, 9, 10 y 11 con $\tilde{y}_C^{(3,i)} \equiv z^{(3,i)}$, hasta que finalmente (3.78) se corresponde con la primera factorización aproximada por bloques construida localmente por la tarea.

La resolución progresiva paralela finaliza cuando todas las hojas del árbol han ejecutado el algoritmo anterior. Los vectores $z^{(3,i)}$, con $i = 1, \ldots, 4$, se pueden particionar conforme al vector residuo de entrada, $r^{(3,i)}$, obteniendo

$$z^{(3,1)} = \left(\begin{array}{c} z_1^{(3,1)} \\ \hline z_5^{(2,1)} \\ z_7^{(1,1)} \end{array}\right), \; z^{(3,2)} = \left(\begin{array}{c} z_2^{(3,2)} \\ \hline z_5^{(2,1)} \\ z_7^{(1,1)} \end{array}\right), \; z^{(3,3)} = \left(\begin{array}{c} z_3^{(3,3)} \\ \hline z_6^{(2,2)} \\ z_7^{(1,1)} \end{array}\right), \; z^{(3,4)} = \left(\begin{array}{c} z_4^{(3,4)} \\ \hline z_6^{(2,2)} \\ z_7^{(1,1)} \end{array}\right).$$

Si relacionamos estos vectores con el vector z que obtiene el Algoritmo 3.11 al procesar todo el precondicionador (ver (3.52)), es posible observar que cada tarea hoja ha calculado su bloque correspondiente de este vector, $z_i^{(3,i)}$, junto con un duplicado de los bloques correspondientes a sus ancestros. Por lo tanto, en este punto, el algoritmo paralelo ya puede obtener la **Salida** del Algoritmo 3.10, recuperando z a partir de $z^{(3,i)}$, con $i = 1, \ldots, 4$ y, posteriormente, aplicando Ω sobre z.

3.3.3. Diseño y justificación de las estrategias de mapeado y planificación

Los microprocesadores de propósito general actuales poseen un sistema de memoria estructurado jerárquicamente, formado por varios niveles de memoria caché, con el objetivo de explotar la localidad de referencia, y atenuar de esa forma la diferencia significativa entre la velocidad del procesador y la memoria principal. El grado en el que un código es capaz de explotar esta jerarquía depende en gran medida del grado de reutilización de datos inherente a la operación. En el que caso que nos ocupa, todas las operaciones necesarias para aplicar el precondicionador multinivel (ver Apartado 3.2.2) son operaciones limitadas por el acceso a memoria en las que existe poca (o nula) reutilización de datos y, por lo tanto, poco margen para explotar la localidad temporal en la jerarquía de memoria. Este bajo grado de reutilización es inherente al tipo de operación (operaciones básicas de álgebra lineal de primer [109] y segundo nivel [59]), independientemente de que las matrices sean densas o dispersas. Sin embargo, en el caso de las matrices dispersas, se dan factores específicos que aumentan todavía más la presión sobre el sistema de memoria, causando que los códigos que aplican el precondicionador obtengan una fracción reducida (típicamente en el intervalo 10-20 %) del rendimiento aritmético pico[2] del procesador. Por ejemplo, en el caso de la operación $y \leftarrow F^T x$ (requerida en la aplicación del precondicionador multinivel; ver Apartado 3.2.2), y teniendo en cuenta que F^T se almacena en formato CSR, estos factores se pueden agrupar en las tres categorías siguientes:

- *Direccionamiento indirecto/baja relación entre el número de operaciones en coma flotante y accesos a memoria.* La estructura de datos correspondiente a F^T almacena los valores

[2]El rendimiento aritmético pico del procesador se define como el máximo número de operaciones en coma flotante por unidad de tiempo que el procesador es capaz de realizar asumiendo un aprovechamiento óptimo de sus unidades funcionales. Por ejemplo, el rendimiento pico del procesador Itanium 2 utilizado en los experimentos de esta tesis es de 6 Gigaflops/seg.; consultar el Apéndice B.

numéricos de los elementos de F^T y sus identificadores de columna. Esto implica que, durante el cálculo del producto $y \leftarrow F^T x$, los identificadores de columna son explícitamente leídos de memoria, aumentando el número de accesos a memoria, el consumo de ancho de banda y la probabilidad de que se produzca "interferencia" entre líneas de caché. De forma más concreta, si $F^T \in \mathbb{R}^{m,n}$, entonces sólo se realizan $2 \cdot nnz(F^T)$ operaciones aritméticas con números reales (en coma flotante), mientras que el número de accesos a números enteros y reales es de $nnz(F^T) + m$, y $m + n + nnz(F^T)$, respectivamente.

- *Baja localidad espacial en el acceso a x*. Los accesos al vector x son discontinuos y, además, pueden ser irregulares, ya que dependen del patrón de elementos no nulos de F^T. Esta propiedad causa típicamente que ninguna de las líneas de caché cargadas en el acceso a x se utilicen íntegramente, reduciendo el margen de mejora de las técnicas tradicionales de precaptación (*prefetching*) de datos.

- *Baja localidad temporal en el acceso a x*. Para minimizar los costes de acceso a memoria, es importante maximizar el número de veces que se reutiliza un dato. Durante el cálculo del producto de una matriz dispersa por un vector, los elementos de F^T son accedidos consecutivamente, por filas, y se usan una única vez, mientras que los elementos de y son accedidos de forma consecutiva, y se reutilizan tantas veces como elementos haya en la fila correspondiente de F^T, lo que resulta óptimo desde el punto de vista de la localidad temporal. Sin embargo, los elementos de x sólo se reutilizan si dos (o más) filas "cercanas" de F^T tienen elementos situados en columnas "cercanas" ya que, al recorrer la segunda fila, es probable que los elementos de x involucrados todavía estén en la caché. En general, este tipo de filas no existen y, por tanto, no hay garantías de reutilización de datos en el acceso a x.

Para el resto de operaciones involucradas en la aplicación del precondicionador, como por ejemplo, el producto de una matriz dispersa por un vector, $y \leftarrow Fx$, y la resolución de sistemas de la forma $LDL^T z = r$, estos factores se pueden agrupar de forma similar.

En el caso paralelo, las decisiones tomadas por las estrategias de mapeado y planificación de tareas tienen, a causa de los factores mencionados previamente, un impacto significativo sobre el rendimiento de la aplicación paralela del precondicionador. La aplicación del precondicionador se realiza en un multiprocesador de memoria compartida en el que cada microprocesador incorpora su propia jerarquía de caché y, además, los datos asociados a las tareas en las que se descompone la resolución regresiva y progresiva se encuentran "ubicados" en el sistema de memoria conforme a la asignación de tareas a procesos resultante del Algoritmo 2.21. Por tanto, si en el proceso de aplicación del precondicionador no se tiene en cuenta este factor a la hora de mapear tareas a procesos, es posible incurrir en un alto sobrecoste asociado al movimiento de datos entre las cachés de los procesadores y memoria principal. Por ejemplo, si la arquitectura del multiprocesador subyacente es del tipo ccNUMA, y no se tiene en cuenta este factor, es posible que los datos asociados a una tarea se tengan que recuperar desde la memoria correspondiente a un procesador "remoto", aumentando aún más la penalización en el acceso a memoria.

El sobrecoste asociado al movimiento de datos entre las cachés de los procesadores y memoria principal es alto cuando, con el propósito de mejorar la eficiencia/equilibrio de carga, se emplean estrategias dinámicas de planificación similares a las del Algoritmo 2.21, en las que cualquier proceso/procesador puede ejecutar cualquier tarea. A diferencia de lo que ocurre con el cálculo paralelo del precondicionador, operación que presenta un mayor grado de reutilización de datos, la experiencia adquirida con la aplicación paralela del precondicionador revela que, en general, el sobrecoste asociado al movimiento de los datos entre los procesos involucrados en la computación paralela no se puede amortizar con las ventajas que ofrecen las estrategias dinámicas de planificación (equilibrado dinámico de la carga computacional).

Resolución progresiva paralela

Con el objetivo de controlar el sobrecoste asociado al movimiento de los datos entre los procesos, mantenemos el mapeado de tareas a procesos resultante del cálculo paralelo del precondicionador para la resolución progresiva paralela. Asumiendo que la asignación de procesos a procesadores no cambia durante la resolución iterativa del sistema, esta decisión permite mejorar el uso de las cachés de los procesadores y reducir el sobrecoste asociado a la comunicación entre éstos, ya que un proceso dado ejecuta repetidamente las mismas tareas (opera con los mismos datos) cada vez que se aplica el precondicionador (ver línea 9 del Algoritmo 3.5). Aunque esta propiedad se cumple en mayor o menor grado para cualquier mapeado *estático* de tareas a procesos, esperamos que el mapeado que resulta del cálculo del precondicionador logre, además, equilibrar la carga computacional de las tareas hoja, ya que este mapeado se obtuvo dinámicamente con este propósito en el proceso de construcción del precondicionador

La especificación de la resolución progresiva paralela se completa concretando la estrategia de planificación que usa el algoritmo para guiar la ejecución concurrente del árbol conforme a las dependencias entre las tareas. Cuando un determinado proceso queda ocioso, esta estrategia decide qué tarea ejecutará este proceso en siguiente lugar entre las tareas asignadas a éste que actualmente se encuentran en estado *ejecutable*. Para tomar esta decisión, cada proceso tiene en cuenta las siguientes reglas:

1. Las tareas hoja tienen prioridad frente a las tareas con hijos. Esta regla trata de evitar el potencial desequilibrio de carga causado por la ejecución tardía de las hojas, que concentran gran parte del coste computacional de la resolución progresiva paralela.

2. Entre las tareas hoja, tiene prioridad aquella que, *potencialmente*, concentre mayor coste computacional. Para tomar esta decisión, se emplea el número de operaciones en coma flotante como una estimación del coste computacional de la tarea. (El número de operaciones en coma flotante se obtiene a partir del número de elementos no nulos de los factores correspondientes a la factorización multinivel calculada localmente por la tarea; ver (2.124).)

3. Entre las tareas con hijos, el proceso escoge, en primer lugar, la tarea que lleve más tiempo en estado ejecutable.

El Algoritmo 3.12 implementa la estrategia de planificación de tareas basada en las reglas anteriores, y combina los bloques básicos presentados en el Apartado 3.3.2 para la resolución progresiva paralela. Cada proceso, con identificador pid, mantiene una cola local, Q_{pid}, que contiene *tareas ejecutables asignadas a éste*, es decir, tareas con sus dependencias resueltas e identificador tid en el conjunto $\{\mathsf{tid} \in T : \mathsf{map}[\mathsf{tid}] = \mathsf{pid}\}$, donde map contiene la asignación de tareas a procesos resultante del cálculo del precondicionador (ver línea 9 del Algoritmo 2.21). La ejecución del algoritmo paralelo comienza con la creación de un grupo de "procesos"[3] (línea 2). Cada proceso inicializa su cola local (línea 4) con las hojas del árbol asignadas a éste en orden decreciente con respecto al número de operaciones en coma flotante que realizan las tareas (segunda regla de planificación). Como resultado de esta inicialización, la tarea hoja que realiza más operaciones se sitúa en la cabeza de Q_{pid} y, la que realiza menos, en su cola. Posteriormente, los procesos monitorizan su cola local a la espera de nuevas tareas ejecutables (bucle **repetir**), y entran en el segundo bucle (**mientras**) siempre que queden tareas en Q_{pid} por ejecutar. La línea 8 construye el residuo de entrada de una tarea, y el núcleo computacional del algoritmo se representa a través de la llamada a la rutina resolución_progresiva_local (línea 9). Cuando un proceso termina la ejecución de una tarea, comprueba además si se ha completado la ejecución

[3] En este contexto utilizamos el término "procesos" como entes abstractos encargados de explotar el paralelismo de tareas. Sin embargo, es necesario clarificar que estos procesos son realmente hebras (hilos de ejecución) con el fin de aprovechar de forma más eficiente la arquitectura multiprocesador de destino mediante el modelo de programación de variables compartidas.

de la tarea hermana y, en dicho caso, inserta la tarea padre en la cola del proceso a cargo de la tarea padre, es decir, en $Q_{\mathsf{map[padre(tid)]}}$ (línea 12).

Algoritmo 3.12: Resolución progresiva paralela.

```
1  marcar todas las tareas de T como no ejecutadas
2  Inicio región paralela                                   ▷ crear grupo de procesos
3      pid ← obtener_pid()                                  ▷ obtener identificador de proceso
       ▷ inicializar Q_pid con las tareas hoja de T a cargo de pid en orden decreciente por #operaciones
4      Q_pid ← { tid ∈ hojas(T) : map[tid] = pid }
5      repetir
6          mientras Q_pid no esté vacía hacer
7              tid ← desencolar(Q_pid)                      ▷ extraer tarea de la cabeza de Q_pid
8              construir_residuo_local(tid)                 ▷ construir residuo correspondiente a tid
9              resolución_progresiva_local(tid)             ▷ calcular resolución progresiva correspondiente a tid
10             marcar tid como ejecutada
11             si se han resuelto las dependencias de padre(tid) entonces
12                 encolar(padre(tid), Q_map[padre(tid)])   ▷ insertar padre en la cola de Q_map[padre(tid)]
13             fin si
14         fin mientras
15     hasta todas las tareas ejecutadas
16 Fin región paralela
```

Resolución regresiva paralela

El mapeado de tareas a procesos resultante del cálculo paralelo del precondicionador también se mantiene, por la misma causa, para la resolución regresiva paralela. Asumiendo que la asignación de procesos a procesadores no cambia durante la resolución iterativa del sistema, cada proceso (procesador) ejecuta repetidamente las mismas tareas (opera con los mismos datos) cada vez que se aplica el precondicionador. Esta decisión permite mejorar el uso de las cachés de los procesadores teniendo en cuenta que: (1) la resolución regresiva ejecuta en primer lugar las tareas que la resolución progresiva ejecutó en último lugar; y (2) las tareas de ambos algoritmos comparten las mismas estructuras de datos. En lo que respecta a la planificación de las tareas del árbol, cada proceso tiene en cuenta las siguientes reglas:

1. *Las tareas con hijos tienen prioridad frente a las hojas.* Las dependencias de las tareas se resuelven en orden inverso en el caso de la resolución regresiva paralela (desde la raíz hasta las hojas) y, por tanto, si un proceso priorizase la ejecución de una hoja frente a una tarea con hijos, entonces se retrasaría la ejecución de esta última y *de todos sus descendientes*. Esta situación puede causar tiempos de espera significativos para los procesos que se encargan de ejecutar sus descendientes, ya que las hojas concentran gran parte de la carga computacional del algoritmo.

2. Entre las tareas con hijos, el proceso escoge la que lleve más tiempo en estado ejecutable.

3. Entre las tareas hoja, el proceso escoge la que lleve más tiempo en estado ejecutable.

El Algoritmo 3.13 incorpora la estrategia de planificación de tareas basada en las reglas anteriores, y combina los bloques básicos presentados en el Apartado 3.3.2 para la resolución regresiva paralela. Cada proceso, con identificador pid, mantiene una cola con prioridad local, H_{pid}, que *sólo contiene tareas ejecutables asignadas a este proceso*. La cola con prioridad gestiona internamente las prioridades entre dichas tareas de tal forma que, cuando un proceso queda

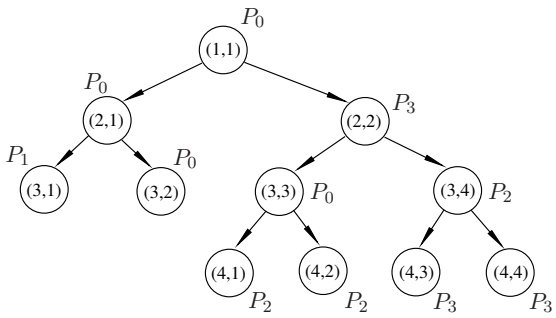
ocioso y solicita una nueva tarea ejecutable, la estructura "le proporciona" la tarea más prioritaria. La ejecución del algoritmo paralelo comienza con la creación de un grupo de procesos (línea 2). En la línea 4, el proceso a cargo de la raíz del árbol inicializa la cola con prioridad local. Posteriormente, los procesos monitorizan su cola con prioridad a la espera de nuevas tareas ejecutables (bucle **repetir**), y entran en el bucle **mientras** siempre que queden tareas en H_{pid} por ejecutar. La línea 8 construye el vector $\tilde{y}_C$ de entrada a una tarea a partir de su padre conforme a la Figura 3.2, y el núcleo computacional del algoritmo se representa a través de la llamada a la rutina resolución_regresiva_local (línea 9). Cuando un proceso completa una tarea, inserta cada uno de sus hijos en las colas con prioridad de los procesos a cargo de éstos, es decir, en $H_{\mathsf{map[tid\_hijo]}}$ (línea 12), donde tid_hijo es el identificador de tarea correspondiente a uno de sus hijos.

Algoritmo 3.13: Resolución regresiva paralela. Mapeado estático.

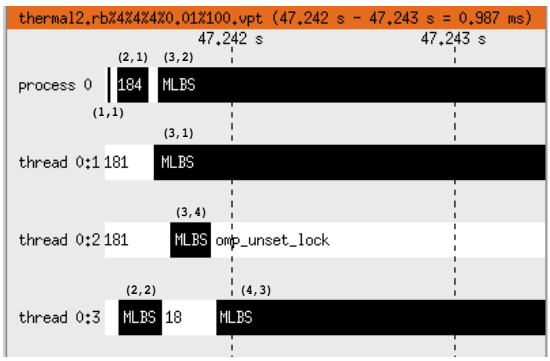
```
1  marcar todas las tareas de T como no ejecutadas
2  Inicio región paralela                                  ▷ crear grupo de procesos
3      pid ← obtener_pid()
       ▷ inicializar H_pid con la tarea raíz de T si ésta está a cargo de pid
4      H_pid ← { tid : tid = raíz(T) ∧ map[tid] = pid }
5      repetir
6          mientras H_pid no esté vacía hacer
7              tid ← extraer(H_pid)                        ▷ extraer tarea más prioritaria de H_pid
8              construir_ỹ_C_local(tid)                    ▷ construir vector ỹ_C correspondiente a tid
9              resolución_regresiva_local(tid)             ▷ calcular resolución regresiva correspondiente a tid
10             marcar tid como ejecutada
11             para tid_hijo ∈ hijos(tid) hacer
12                 insertar(tid_hijo, H_map[tid_hijo])     ▷ insertar hijo en H_map[tid_hijo]
13             fin para
14         fin mientras
15     hasta todas las tareas ejecutadas
16 Fin región paralela
```

Cuando un proceso queda ocioso y solicita una nueva tarea, su cola con prioridad "le proporciona" la tarea más prioritaria. En el caso de que sólo haya tareas hoja en esta estructura, el proceso inicia la ejecución de una de ellas, y no tiene en cuenta que puede haber varias tareas con hijos mapeadas a éste que no están todavía en su cola con prioridad, es decir, que no se pueden ejecutar en el estado actual de la computación paralela (porque todavía no se han resuelto sus dependencias). Por lo tanto, si el proceso sigue ejecutando la tarea hoja cuando se resuelven las dependencias de una (o varias) de las tareas con hijos mapeadas a este proceso, entonces esta tarea con hijos se queda a la espera en la cola con prioridad local del proceso, por lo que se retrasa su ejecución y la de todos sus descendientes. Este retraso puede causar tiempos de espera significativos para los procesos que se encargan de ejecutar sus descendientes, ya que las hojas concentran gran parte de la carga computacional del algoritmo. La Figura 3.3 ilustra un escenario real de planificación del proceso de resolución regresiva paralela en el que el rendimiento del Algoritmo 3.13 se ve *severamente* afectado por este retraso. La Figura 3.3 (a) muestra el mapeado de tareas (etiquetas interiores de los nodos) a procesos (etiquetas exteriores de los nodos) resultante del Algoritmo 2.21, y la Figura 3.3 (b) es un diagrama temporal del primer *milisegundo* de ejecución de nuestra implementación del Algoritmo 3.13 en el multiprocesador SGI Altix 350 (consultar el Apéndice B). Este diagrama muestra las actividades que realizan los procesos (hebras) a lo largo de un eje horizontal de tiempo. En los intervalos de color negro, los procesos ejecutan tareas y, en los de color blanco, se sincronizan o están a la *espera* de nuevas tareas ejecutables. El diagrama sólo muestra el primer milisegundo de ejecución ya que, para

este caso particular, este intervalo es suficiente para ilustrar la espera causada por las decisiones de planificación que toma el Algoritmo 3.13. Como se puede observar en la Figura 3.3 (b), P_3 completa la ejecución de $(2,2)$ cuando P_0 ya ha empezado a ejecutar $(3,2)$ y, por lo tanto, $(3,3)$ permanece en la cola con prioridad de P_0 esperando a que este proceso complete la ejecución de $(3,2)$. Como consecuencia, cuando P_2 completa $(3,4)$, queda ocioso a la espera de que se resuelvan las dependencias de $(4,1)$ y $(4,2)$.



(a)



(b)

Figura 3.3: Algoritmo 3.13 con $p = 4$ procesos. (a) Mapeado de tareas a procesos. (b) Diagrama temporal del primer milisegundo de ejecución.

Para reducir el retraso causado por la situación discutida previamente, hemos optado por permitir, en tiempo de ejecución, ajustes en el mapeado inicial de tareas con hijos a procesos. El Algoritmo 3.14 controla el tipo de tarea que los procesos están ejecutando en cada momento y utiliza esta información para planificar la ejecución de las tareas del árbol. Cuando un proceso, con identificador pid, completa una tarea con hijos (línea 10), comprueba para cada hijo (tid_hijo) que tenga descendientes, el tipo de tarea que está ejecutando el proceso correspondiente (map[tid_hijo]), y si éste último está ejecutando actualmente una hoja, *entonces pid se hace cargo insertándolo en su cola con prioridad local* (línea 13). El propósito de realizar este ajuste en el mapeado de tareas a procesos es que la tarea tid_hijo tenga que esperar menos tiempo en la cola de pid que en la del proceso que inicialmente estaba a cargo de su ejecución (map[tid_hijo]), aunque no existen garantías de que esto se cumpla. Esperamos que la penalización en la que se incurre en ese caso no sea muy alta ya que, una vez realizado el ajuste, la tarea sólo tiene que esperar a que se completen las tareas con hijos que en ese momento hay en la cola con prioridad local del proceso, tareas que concentran un coste computacional relativamente bajo.

La Figura 3.4 (a) muestra el diagrama temporal correspondiente al primer *milisegundo* de la ejecución del Algoritmo 3.14 cuando éste parte del mismo mapeado de tareas a procesos; ver Figura 3.3 (a). Cuando P_3 completa $(2,2)$, comprueba si el proceso (inicialmente) a cargo de $(3,3)$, es decir, P_0, está actualmente ejecutando una tarea hoja, y como está ejecutando $(3,2)$, entonces P_3 se hace cargo de $(3,3)$. Este ajuste permite ejecutar rápidamente la tarea $(3,3)$, y resolver de esa forma las dependencias de $(4,1)$ y $(4,2)$. Por lo tanto, a diferencia de la Figura 3.3 (b), P_2 ya no se queda ocioso a la espera de tareas ejecutables. La Figura 3.4 (b) ilustra el mapeado de tareas a procesos que resulta tras efectuar este ajuste.

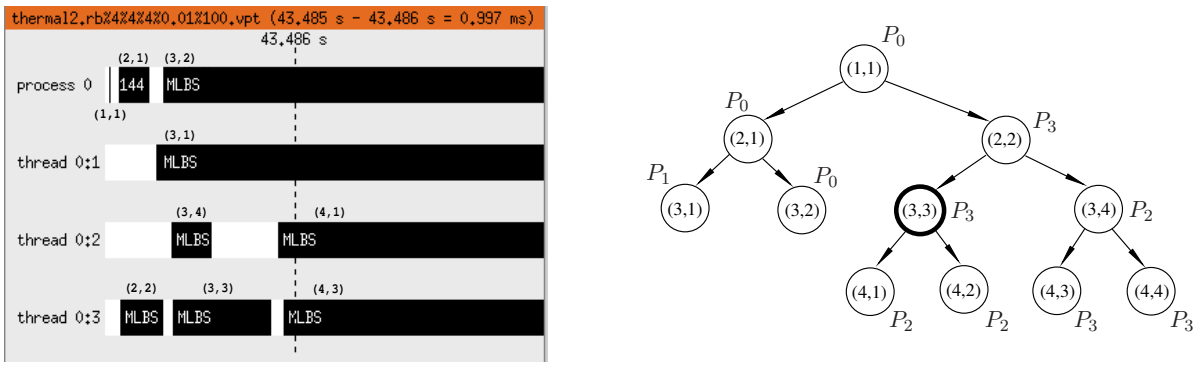
3.4. El método PCG paralelo

El punto de partida para el diseño del método PCG paralelo es el resolutor "secuencial"[4] que se muestra en el Algoritmo 3.15. Este algoritmo se obtiene aplicando el método PCG (ver

[4]Este resolutor no explota el paralelismo de tareas, de ahí el término resolutor "secuencial".

Algoritmo 3.14: Resolución regresiva paralela. Mapeado estático con ajustes.

```
1  marcar todas las tareas de T como no ejecutadas
2  Inicio región paralela                                   ▷ crear grupo de procesos
3      pid ← obtener_pid()
       ▷ inicializar H_pid con la tarea raíz de T si ésta está a cargo de pid
4      H_pid ← { tid : tid = raíz(T) ∧ map[tid] = pid }
5      repetir
6          mientras H_pid no esté vacía hacer
7              tid ← extraer(H_pid)                         ▷ extraer tarea más prioritaria de H_pid
8              construir_ỹ_C_local(tid)                     ▷ construir vector ỹ_C correspondiente a tid
9              resolución_regresiva_local(tid)              ▷ calcular resolución regresiva correspondiente a tid
10             marcar tid como ejecutada
11             para tid_hijo ∈ hijos(tid) hacer
12                 si hijos(tid_hijo) ≠ ∅ y proceso map[tid_hijo] ejecutando hoja entonces
13                     insertar(tid_hijo, H_pid)            ▷ proceso pid ahora a cargo de tid_hijo
14                 si no
15                     insertar(tid_hijo, H_map[tid_hijo])  ▷ insertar tid_hijo en H_map[tid_hijo]
16                 fin si
17             fin para
18         fin mientras
19     hasta todas las tareas ejecutadas
20 Fin región paralela
```



(a) (b)

Figura 3.4: Algoritmo 3.14 con $p = 4$ procesos. (a) Diagrama temporal del primer milisegundo. (b) Mapeado de tareas a procesos usado en su ejecución.

Algoritmo 3.5) al sistema transformado

$$\tilde{A}\tilde{x} = \tilde{b} \;\equiv\; (\Omega^T A \Omega)(\Omega^T x) = (\Omega^T b), \tag{3.79}$$

con el precondicionador M construido por el Algoritmo 2.19. En el Algoritmo 3.15, los superíndices de los vectores y escalares se han omitido para simplificar la notación y, además, se ha añadido un nuevo vector $\tilde{q}$ que almacena el producto de $\tilde{A}$ por p, y tres nuevos escalares, σ_1, σ_2 y δ que contienen, en la iteración i-ésima, los productos escalares $(\tilde{r}^{(i)})^T\tilde{z}^{(i)}$, $(\tilde{r}^{(i+1)})^T\tilde{z}^{(i+1)}$ y $(\tilde{p}^{(i)})^T A\tilde{p}^{(i)}$, respectivamente. La ordenación de los componentes de los vectores $\tilde{x}$, $\tilde{p}$, $\tilde{r}$, $\tilde{q}$ y $\tilde{z}$ coincide con la de las incógnitas del sistema transformado (3.79) y, por lo tanto, cada vez que el Algoritmo 3.10 aplica M (líneas 4 y 12), ya no es necesario recuperar esta ordenación en el residuo $\tilde{r}$ y en el residuo precondicionado $\tilde{z}$ (ver **Entrada** y **Salida** del Algoritmo 3.10). Cuando el método converge, la solución del sistema original $Ax = b$ se obtiene, en la línea 19, aplicando la matriz de permutación Ω a $\tilde{x}$.

Algoritmo 3.15: Método iterativo de resolución de $Ax = b$. El método se deriva aplicando el Algoritmo 3.5 al sistema (3.79) con el precondicionador M construido por el Algoritmo 2.19.

```
1  Fijar una solución inicial x̃
2  q̃ ← Ãx̃
3  r̃ ← b̃ − q̃
4  Resolver Mz̃ = r̃ mediante el Algoritmo 3.10
5  σ1 ← r̃^T z̃
6  p̃ ← z̃
7  para i = 0, 1, ..., hasta convergencia hacer
8      δ ← p̃^T q̃
9      α ← σ1/δ
10     x̃ ← x̃ + αp̃
11     r̃ ← r̃ − αq̃
12     Resolver Mz̃ = r̃ mediante el Algoritmo 3.10
13     σ2 ← r̃^T z̃
14     β ← σ2/σ1
15     σ1 ← σ2
16     p̃ ← z̃ + βp̃
17     q̃ ← Ãp̃
18 fin para
19 x ← Ωx̃
```

Al igual que las operaciones involucradas en la aplicación del precondicionador, el resto de operaciones básicas del Algoritmo 3.15 son operaciones limitadas por el acceso a memoria en las que existe muy poca (o nula) reutilización de datos y, por lo tanto, el acceso a memoria es un factor clave a considerar en el diseño de su paralelización. Con el objetivo de mejorar la localidad y, en consecuencia, reducir el sobrecoste asociado al tráfico de datos en el sistema de memoria del multiprocesador, descomponemos todos los vectores involucrados en el método conforme a los vectores $\tilde{r}$ y $\tilde{z}$ en la aplicación paralela del precondicionador; ver Apartado 3.3.2. En particular, descomponemos $\tilde{b}$ y $\tilde{q}$ conforme a $\tilde{r}$, y $\tilde{x}$ y $\tilde{p}$ conforme a $\tilde{z}$. Consideremos, por ejemplo, el árbol

de la Figura 3.1; $\tilde{q}$ se descompone en los siguientes sumandos, uno por cada tarea hoja,

$$\tilde{q} = \begin{pmatrix} \tilde{q}_1^{(3,1)} \\ \tilde{q}_2^{(3,2)} \\ \tilde{q}_3^{(3,3)} \\ \tilde{q}_4^{(3,4)} \\ \hline \tilde{q}_5^{(2,1)} \\ \tilde{q}_6^{(2,2)} \\ \hline\hline \tilde{q}_7^{(1,1)} \end{pmatrix} = \begin{pmatrix} \tilde{q}_1^{(3,1)} \\ 0 \\ 0 \\ 0 \\ \hline \tilde{q}_5^{(3,1)} \\ 0 \\ \hline\hline \tilde{q}_7^{(3,1)} \end{pmatrix} + \begin{pmatrix} 0 \\ \tilde{q}_2^{(3,2)} \\ 0 \\ 0 \\ \hline \tilde{q}_5^{(3,2)} \\ 0 \\ \hline\hline \tilde{q}_7^{(3,2)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \tilde{q}_3^{(3,3)} \\ 0 \\ \hline 0 \\ \tilde{q}_6^{(3,3)} \\ \hline\hline \tilde{q}_7^{(3,3)} \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \tilde{q}_4^{(3,4)} \\ \hline 0 \\ \tilde{q}_6^{(3,4)} \\ \hline\hline \tilde{q}_7^{(3,4)} \end{pmatrix} \tag{3.80}$$

$$= \tilde{q}^{(3,1)} + \tilde{q}^{(3,2)} + \tilde{q}^{(3,3)} + \tilde{q}^{(3,4)},$$

donde los bloques locales de una tarea hoja correspondientes a sus ancestros (por ejemplo, $\tilde{q}_5^{(3,1)}$ y $\tilde{q}_7^{(3,1)}$ en el caso de $(3,1)$) contienen *contribuciones* a las entradas correspondientes de $\tilde{q}$, y $\tilde{p}$ se descompone en

$$\tilde{p} = \begin{pmatrix} \tilde{p}_1^{(3,1)} \\ \tilde{p}_2^{(3,2)} \\ \tilde{p}_3^{(3,3)} \\ \tilde{p}_4^{(3,4)} \\ \hline \tilde{p}_5^{(2,1)} \\ \tilde{p}_6^{(2,2)} \\ \hline\hline \tilde{p}_7^{(1,1)} \end{pmatrix} ; \underbrace{\begin{pmatrix} \tilde{p}_1^{(3,1)} \\ 0 \\ 0 \\ 0 \\ \hline \tilde{p}_5^{(2,1)} \\ 0 \\ \hline\hline \tilde{p}_7^{(1,1)} \end{pmatrix}}_{\tilde{p}^{(3,1)}}, \underbrace{\begin{pmatrix} 0 \\ \tilde{p}_2^{(3,2)} \\ 0 \\ 0 \\ \hline \tilde{p}_5^{(2,1)} \\ 0 \\ \hline\hline \tilde{p}_7^{(1,1)} \end{pmatrix}}_{\tilde{p}^{(3,2)}}, \underbrace{\begin{pmatrix} 0 \\ 0 \\ \tilde{p}_3^{(3,3)} \\ 0 \\ \hline 0 \\ \tilde{p}_6^{(2,2)} \\ \hline\hline \tilde{p}_7^{(1,1)} \end{pmatrix}}_{\tilde{p}^{(3,3)}}, \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ \tilde{p}_4^{(3,4)} \\ \hline 0 \\ \tilde{p}_6^{(2,2)} \\ \hline\hline \tilde{p}_7^{(1,2)} \end{pmatrix}}_{\tilde{p}^{(3,4)}}, \tag{3.81}$$

(3.82)

donde los bloques locales de una tarea hoja correspondientes a sus ancestros (por ejemplo, $\tilde{p}_5^{(2,1)}$ y $\tilde{p}_7^{(1,1)}$ en el caso de $(3,1)$) contienen *duplicados* de las entradas correspondientes de $\tilde{p}$. Al iniciarse la ejecución del método PCG paralelo, los procesos descomponen *explícitamente* los vectores $\tilde{x}$, $\tilde{p}$, $\tilde{r}$, $\tilde{q}$, y $\tilde{z}$ y, posteriormente, ejecutan las mismas tareas (operan con las mismas estructuras de datos locales) durante todo el proceso de resolución iterativa del sistema, mejorando el uso de las cachés de los procesadores y reduciendo la comunicación entre éstos.

En esta sección presentamos cómo aprovechar el paralelismo disponible en las operaciones del Algoritmo 3.15 teniendo en cuenta las consideraciones anteriores. Cabe recordar que el Apartado 3.3.2 se centró en la aplicación del precondicionador M (líneas 4 y 12). Los Apartados 3.4.1, 3.4.2, y 3.4.3 revisan el resto de operaciones básicas del método, es decir, las actualizaciones de vectores (líneas 3, 10, 11 y 16), los productos escalares (líneas 5, 8 y 13), y el producto de una matriz dispersa por un vector (líneas 2 y 17). A continuación, el Apartado 3.4.4 combina los distintos bloques básicos para derivar finalmente el método PCG paralelo y, por último, el Apartado 3.4.5 evalúa el rendimiento del método cuando se aplica para resolver los sistemas de ecuaciones que surgen en el contexto de la EDP de difusión con coeficientes discontinuos.

3.4.1. Actualizaciones de vectores

En el Algoritmo 3.15, *todos* los operandos de una actualización dada se descomponen conforme a (3.80) (líneas 3 y 11) o (3.81) (líneas 10 y 16). Por lo tanto, para paralelizar las actualizaciones de vectores, cada tarea hoja simplemente actualiza su vector local de forma independiente al resto. Después de esta actualización, los bloques locales de la tarea correspondientes a sus ancestros siguen almacenando *contribuciones* a las entradas del vector actualizado, en el caso

de las líneas 3 y 11, y *duplicados* en el caso de las líneas 10 y 16. Por ejemplo, la tarea $(3,2)$ actualiza $\tilde{r}^{(3,2)}$ localmente como

$$\begin{pmatrix} \tilde{r}_2^{(3,2)} \\ \tilde{r}_5^{(3,2)} \\ \tilde{r}_7^{(3,2)} \end{pmatrix} \leftarrow \begin{pmatrix} \tilde{r}_2^{(3,2)} \\ \tilde{r}_5^{(3,2)} \\ \tilde{r}_7^{(3,2)} \end{pmatrix} - \alpha \begin{pmatrix} \tilde{q}_2^{(3,2)} \\ \tilde{q}_5^{(3,2)} \\ \tilde{q}_7^{(3,2)} \end{pmatrix}, \tag{3.83}$$

y, después de esta actualización, $\tilde{r}_5^{(3,2)}$ y $\tilde{r}_7^{(3,2)}$ siguen almacenando contribuciones a las entradas de $\tilde{r}$.

La actualización paralela de vectores se muestra en el Algoritmo 3.16. Cada proceso, con identificador pid, realiza las computaciones de las tareas hoja con identificador tid en el conjunto $\{\mathsf{tid} \in \mathsf{hojas}(T) : \mathsf{map[tid]} = \mathsf{pid}\}$, donde map contiene la asignación de tareas a procesos resultante del cálculo paralelo del precondicionador (ver línea 9 del Algoritmo 2.21).

Algoritmo 3.16: Actualización paralela de vectores.

```
1 pid ← obtener_pid()                          ▷ obtener identificador de proceso
2 para {tid ∈ hojas(T) : map[tid] = pid} hacer
3     actualización_local(tid)                 ▷ actualizar vector correspondiente a tid
4 fin para
```

3.4.2. Productos escalares

En todos los productos escalares del Algoritmo 3.15, uno de los vectores se descompone conforme a (3.80) y, el otro, conforme a (3.81). Teniendo en cuenta que los bloques locales de las hojas correspondientes a sus ancestros contienen, respectivamente, contribuciones a/duplicados de las entradas de estos vectores, el producto escalar se puede paralelizar en los dos pasos siguientes:

- En el primer paso, las tareas hoja calculan un producto escalar *parcial* de forma local e independiente al resto. Por ejemplo, en el caso del producto escalar de la línea 8, la tarea $(3,2)$ calcula este producto *parcial* como

$$\delta^{(3,2)} \leftarrow \begin{pmatrix} \tilde{p}_2^{(3,2)} & \tilde{p}_5^{(2,1)} & \tilde{p}_7^{(1,1)} \end{pmatrix} \begin{pmatrix} \tilde{q}_2^{(3,2)} \\ \tilde{q}_5^{(3,2)} \\ \tilde{q}_7^{(3,2)} \end{pmatrix} = (\tilde{p}_2^{(3,2)})^T \tilde{q}_2^{(3,2)} + (\tilde{p}_5^{(2,1)})^T \tilde{q}_5^{(3,2)} + (\tilde{p}_7^{(1,1)})^T \tilde{q}_7^{(3,2)}.$$

- En el segundo paso, los procesos a cargo de las tareas hoja se sincronizan para acumular, en una operación de reducción, los productos parciales calculados en el primer paso. Por ejemplo, el resultado del producto escalar de la línea 8 se obtiene como

$$\delta \leftarrow \delta^{(3,1)} + \delta^{(3,2)} + \delta^{(3,3)} + \delta^{(3,4)}. \tag{3.84}$$

El producto escalar paralelo se muestra en el Algoritmo 3.16. En el bucle de la línea 3, cada proceso, con identificador pid, calcula y acumula en α_{pid} los productos parciales de las tareas hoja con identificador tid en el conjunto $\{\mathsf{tid} \in \mathsf{hojas}(T) : \mathsf{map[tid]} = \mathsf{pid}\}$. Posteriormente, los procesos se sincronizan en la línea 6 para acumular estos productos parciales mediante una operación de reducción, obteniendo de esa forma el resultado final, $\alpha \leftarrow x^T y$.

Algoritmo 3.17: Producto escalar paralelo.

```
1 pid ← obtener_pid()                                   ▷ obtener identificador de proceso
2 α_pid ← 0
3 para {tid ∈ hojas(T) : map[tid] = pid} hacer
4     α_pid ← α_pid + producto_escalar_parcial(tid)     ▷ calcular producto parcial correspondiente a tid
5 fin para
6 α ← reducción(α_pid, +)                               ▷ reducción de los productos parciales
```

3.4.3. Producto de una matriz dispersa por un vector

En los productos $\tilde{q} \leftarrow \tilde{A}\tilde{x}$ (línea 2) y $\tilde{q} \leftarrow \tilde{A}\tilde{p}$ (línea 17), $\tilde{q}$ se descompone conforme a (3.80), $\tilde{A}$ conforme a la Figura 2.19, y $\tilde{x}$ y $\tilde{p}$ conforme a (3.81). Por lo tanto, para paralelizar estas operaciones, cada tarea hoja, $(3,i)$, con $i = 1,\ldots,4$, realiza simplemente el producto $\tilde{q}^{(3,i)} \leftarrow \tilde{A}^{(3,i)}\tilde{x}^{(3,i)}$ o $\tilde{q}^{(3,i)} \leftarrow \tilde{A}^{(3,i)}\tilde{p}^{(3,i)}$, de forma independiente al resto. Después de calcular este producto, los bloques locales de la tarea correspondientes a sus ancestros siguen almacenando contribuciones a las entradas de $\tilde{q}$. Por ejemplo, en el caso de la línea 17, la tarea $(3,2)$ efectúa locamente el siguiente producto

$$\begin{pmatrix} \tilde{q}_2^{(3,2)} \\ \tilde{q}_5^{(3,2)} \\ \tilde{q}_7^{(3,2)} \end{pmatrix} \leftarrow \begin{pmatrix} A_{22}^{(3,2)} & (A_{52}^{(3,2)})^T & (A_{72}^{(3,2)})^T \\ A_{52}^{(3,2)} & A_{55}^{(3,2)} & (A_{75}^{(3,2)})^T \\ A_{72}^{(3,2)} & A_{75}^{(3,2)} & A_{77}^{(3,2)} \end{pmatrix} \begin{pmatrix} \tilde{p}_2^{(3,2)} \\ \tilde{p}_5^{(2,1)} \\ \tilde{p}_7^{(1,1)} \end{pmatrix},$$

donde $\tilde{q}_5^{(3,2)}$ y $\tilde{q}_7^{(3,2)}$ siguen almacenando contribuciones a las entradas de $\tilde{q}$.

El producto paralelo de una matriz dispersa por un vector se muestra en el Algoritmo 3.18. Cada proceso, con identificador pid, simplemente realiza las computaciones locales de las tareas hoja con identificador tid en el conjunto $\{\mathsf{tid} \in \mathsf{hojas}(T) : \mathsf{map[tid]} = \mathsf{pid}\}$.

Algoritmo 3.18: Producto paralelo de una matriz dispersa por un vector.

```
1 pid ← obtener_pid()                                   ▷ obtener identificador de proceso
2 para {tid ∈ hojas(T) : map[tid] = pid} hacer
3     producto_matriz_vector_disperso_local(tid)
4 fin para
```

3.4.4. El método paralelo de los gradientes conjugados precondicionados

El método PCG paralelo se muestra en el Algoritmo 3.19. Este algoritmo combina los bloques básicos de los Apartados 3.3.2, 3.4.1, 3.4.2 y 3.4.3 para calcular, en paralelo, la aplicación del precondicionador M (líneas 6 y 14), las actualizaciones de vectores (líneas 5, 12, 13 y 18), los productos escalares (líneas 7, 10 y 15) y el producto de una matriz dispersa por un vector (líneas 4 y 19). El enfoque de paralelización escogido para las operaciones básicas del método PCG permite eliminar la región paralela de los Algoritmos 3.12, 3.13 y 3.14, e introducir una sola región paralela en el Algoritmo 3.19, reduciendo a uno el número de operaciones de creación (*fork*) y sincronización (*join*) de procesos (hebras). La ejecución paralela del algoritmo comienza, en la línea 2, creando un grupo de procesos. En la línea 3, los vectores $\tilde{b}$, $\tilde{r}$ y $\tilde{q}$ se descomponen explícitamente conforme a (3.80), y $\tilde{x}$, $\tilde{p}$ y $\tilde{z}$ conforme a (3.81) y, posteriormente, los procesos operan con las estructuras de datos locales de las tareas durante todo el proceso de resolución iterativa del sistema. Los escalares σ_1, σ_2, δ, α y β se almacenan en variables compartidas. Por simplicidad, se han omitido en el Algoritmo 3.19 los detalles referentes al acceso concurrente seguro a estas variables compartidas.

Algoritmo 3.19: Método iterativo paralelo de resolución de $Ax = b$.

1 Fijar una solución inicial $\tilde{x}$
2 **Inicio región paralela** ▷ crear grupo de procesos
3 Descomponer $\tilde{b}$, $\tilde{r}$, $\tilde{q}$, $\tilde{x}$, $\tilde{p}$ y $\tilde{z}$
4 Calcular $\tilde{q} \leftarrow \tilde{A}\tilde{x}$ mediante el Algoritmo 3.18
5 Calcular $\tilde{r} \leftarrow \tilde{b} - \tilde{q}$ mediante el Algoritmo 3.16
6 Resolver $M\tilde{z} = \tilde{r}$ mediante el Algoritmos 3.12 y (3.13 ó 3.14)
7 Calcular $\sigma_1 \leftarrow \tilde{r}^T\tilde{z}$ mediante el Algoritmo 3.17
8 Calcular $\tilde{p} \leftarrow \tilde{z}$
9 **para** $i = 0, 1, \ldots,$ *hasta convergencia* **hacer**
10 Calcular $\delta \leftarrow \tilde{p}^T\tilde{q}$ mediante el Algoritmo 3.17
11 Calcular $\alpha \leftarrow \frac{\sigma_1}{\delta}$
12 Calcular $\tilde{x} \leftarrow \tilde{x} + \alpha\tilde{p}$ mediante el Algoritmo 3.16
13 Calcular $\tilde{r} \leftarrow \tilde{r} - \alpha\tilde{q}$ mediante el Algoritmo 3.16
14 Resolver $M\tilde{z} = \tilde{r}$ mediante el Algoritmos 3.12 y (3.13 ó 3.14)
15 Calcular $\sigma_2 \leftarrow \tilde{r}^T\tilde{z}$ mediante el Algoritmo 3.17
16 $\beta \leftarrow \frac{\sigma_2}{\sigma_1}$
17 $\sigma_1 \leftarrow \sigma_2$
18 Calcular $\tilde{p} \leftarrow \tilde{z} + \beta\tilde{p}$ mediante el Algoritmo 3.16
19 Calcular $\tilde{q} \leftarrow \tilde{A}\tilde{p}$ mediante el Algoritmo 3.18
20 **fin para**
21 Reconstruir $\tilde{x}$ a partir de los vectores locales de las tareas hoja
22 **Fin región paralela**
23 $x \leftarrow \Omega\tilde{x}$

3.4.5. Ejemplo de aplicación a la EDP de difusión con coeficientes discontinuos

En este apartado evaluamos el rendimiento del Algoritmo 3.19 cuando se aplica para resolver, en el multiprocesador SGI Altix 350 (consultar el Apéndice B), sistemas de ecuaciones lineales que surgen en el contexto de la discretización de la EDP de difusión en estado estacionario con coeficientes discontinuos. En concreto, consideramos la resolución de los cuatro sistemas de ecuaciones de la Figura 2.20, con los precondicionadores que el Algoritmo 2.21 construye para estos sistemas (consultar el Apartado 2.3.5). La evaluación se centrará en la aplicación del precondicionador y en el producto de una matriz dispersa por un vector, ya que estas operaciones básicas concentran la mayor parte del coste computacional del Algoritmo 3.19; los resultados obtenidos con la versión secuencial de la biblioteca ILUPACK en un microprocesador Itanium 2 (consultar el Apéndice B) revelan que, para cualquiera de los cuatro sistemas considerados, estas operaciones concentran, respectivamente, 89.3 % y 8.5 % del coste computacional del método PCG.

Descripción y justificación de los experimentos

Según lo especificado en el Apartado 3.3.3, la asignación de tareas a procesos que resulta del proceso de construcción del precondicionador, se mantiene para los Algoritmos 3.12, 3.13 y 3.14. Por tanto, el rendimiento de estos algoritmos se ve afectado por esta asignación, que puede cambiar de una ejecución a otra distinta del Algoritmo 2.21 (debido a que las tareas se asignan dinámicamente a los procesos durante el proceso de construcción del precondicionador). Para estudiar la influencia del mapeado inicial de tareas a procesos en el rendimiento de la resolución regresiva y progresiva paralela, y generar distintos escenarios para comparar los Algoritmos 3.13 y 3.14, ejecutamos 50 veces la terna formada por: (1) Algoritmo 2.21; (2) Algoritmo 3.19 utili-

zando los Algoritmos 3.12 y 3.13; y (3) Algoritmo 3.19 utilizando los Algoritmos 3.12 y 3.14. En cada una de las ejecuciones, el mapeado de tareas a procesos que resulta de (1), se mantiene para (2) y (3). Midiendo *el tiempo medio por iteración* que cuesta aplicar el precondicionador en (2) y (3), constituimos una muestra estadística de 50 resultados para (2), y otra de 50 para (3).

Rendimiento de la resolución regresiva y progresiva paralela

Las Tablas 3.1, 3.2, 3.3 y 3.4 recogen únicamente los resultados de la muestra correspondiente a (3)[5]. El número de procesos involucrados en la computación paralela se especifica a través de la columna etiquetada como p. Los resultados mostrados para $p = 1$ se obtuvieron con la implementación del método PCG que incorpora la biblioteca numérica ILUPACK. La altura (número de niveles) y el número de hojas del árbol se fijaron durante la fase de particionado como una función del valor del parámetro f; ver Algoritmo 2.20. El experimento explora diferentes valores del parámetro f en el intervalo $[p, 3p]$. En concreto, estos valores se obtienen multiplicando el número de procesos por el valor indicado en la columna etiquetada como f. Las columnas etiquetadas como "T_p (ms)" muestran el tiempo de ejecución paralelo en milisegundos de la resolución progresiva y regresiva en la *mediana* de la muestra; y las etiquetadas como "cv (%)", el *coeficiente de variación* que se obtiene como el cociente entre la media aritmética y la desviación típica de la suma de los costes computacionales de las tareas que los Algoritmos 3.12 y 3.14 asignan a cada proceso en la ejecución correspondiente a la mediana de la muestra. Por lo tanto, un valor más pequeño del coeficiente de variación indica una distribución más homogénea de la carga computacional en la mediana de la muestra. Las columnas etiquetadas como "Acel. (%)" cuantifican la aceleración (valores positivos) o deceleración (valores negativos) relativa en el tiempo de ejecución que se obtiene empleando valores de f en el intervalo $[p, 3p]$, con respecto al que se obtiene con $f = c$. Por último, las columnas etiquetadas como "σ (ms)" contienen la desviación estándar de la muestra formada por los tiempos de ejecución de la resolución progresiva y regresiva en cada una de las 50 ejecuciones de (3). Este indicador mide la dispersión que existe entre los tiempos de ejecución en función del mapeado inicial de tareas a procesos. Es necesario remarcar que, mientras "σ (ms)" mide el grado de homogeneidad de la muestra formada por *todos los tiempos de ejecución*, "cv (%)" mide el grado de homogeneidad de la distribución de la carga correspondiente a *una ejecución concreta de la muestra* (la correspondiente a la mediana).

Antes de comenzar el análisis detallado de los resultados, es importante mencionar que para todas las combinaciones de p y f mostradas en las Tablas 3.1, 3.2, 3.3 y 3.4, el valor de las columnas "σ (ms)" fue relativamente pequeño en relación al valor de "T_p (ms)". Por tanto, el mapeado inicial de tareas a procesos tiene un impacto moderado sobre la dispersión de la muestra formada por los tiempos de ejecución de la resolución progresiva y regresiva en cada una de las 50 ejecuciones de (3).

Para comenzar la discusión de las Tablas 3.1, 3.2, 3.3 y 3.4, nos centramos en los resultados obtenidos con 2 y 4 procesos. Como se puede observar en las tablas, siempre se obtiene una reducción del tiempo de ejecución paralelo escogiendo el valor de f en el intervalo $[p, 3p]$, (los valores de la columna "Acel. (%)" son positivos en todos los casos) y, además, esta reducción aumenta *generalmente* con f. Por ejemplo, en la Tabla 3.1, la aceleración aumenta desde el 3,3 % ($p = 4/f = p \times 1{,}0$) hasta el 27,2 % ($p = 4/f = p \times 2{,}75$) para la resolución progresiva y, desde el 8,7 % hasta el 31,4 % para la regresiva. La justificación a estas observaciones se puede esbozar teniendo en cuenta que, al dividir el cómputo de la resolución regresiva y progresiva en árboles de tareas con mayor número de niveles/hojas, se ponen de manifiesto dos factores favorables con un peso significativo en las prestaciones: en primer lugar, se consigue una distribución más equilibrada de la carga de trabajo (los valores de la columna "cv (%)" generalmente disminuyen); en segundo lugar, el tamaño del espacio de trabajo (*working set size*) de las tareas se reduce, mejorando el uso de las cachés de los procesadores. Resulta interesante remarcar que el peso de

[5]Los resultados de la muestra correspondiente a (2) se tienen en cuenta más adelante, cuando se compara el rendimiento de los Algoritmos 3.13 y 3.14.

		Árbol de tareas		Resolución Progresiva Paralela				Resolución Regresiva Paralela			
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (ms)	σ (ms)	Acel. (%)	cv (%)
1	× 1,00	1	1	344,0	0,3	0,0	0,0	283,3	0,2	0,0	0,0
2	c	2	2	216,8	0,5	0,0	29,3	178,2	0,3	0,0	31,8
2	× 1,00	3	3	207,5	0,3	4,3	34,9	169,7	0,2	4,8	32,9
2	× 1,25	3	4	150,7	0,4	30,5	1,0	124,2	0,5	30,3	1,0
2	× 1,50	3	4	150,5	0,4	30,6	0,7	124,1	0,5	30,4	0,7
2	× 1,75	3	4	150,5	0,4	30,6	0,9	124,3	0,5	30,2	0,7
2	× 2,00	4	6	142,1	0,4	34,5	0,5	117,2	0,4	34,2	0,8
2	× 2,25	4	8	143,3	0,3	33,9	2,2	115,4	0,3	35,2	1,8
2	× 2,50	4	8	142,9	0,2	34,1	2,4	115,9	0,2	35,0	2,0
2	× 2,75	4	8	143,1	0,2	34,0	2,2	115,7	0,3	35,1	1,7
2	× 3,00	4	8	142,9	0,3	34,1	1,9	115,9	0,3	35,0	1,7
4	c	3	4	98,1	0,2	0,0	31,4	83,0	0,2	0,0	35,7
4	× 1,00	4	6	94,9	0,3	3,3	35,2	75,8	0,3	8,7	34,0
4	× 1,25	4	8	96,3	0,2	1,8	25,7	78,6	0,4	5,3	25,9
4	× 1,50	4	8	96,2	0,2	1,9	25,6	78,6	0,5	5,3	26,1
4	× 1,75	4	8	96,2	0,2	1,9	25,7	78,6	0,5	5,3	26,0
4	× 2,00	5	13	76,4	0,2	22,1	8,0	63,2	0,2	23,9	9,6
4	× 2,25	5	14	78,5	0,2	20,0	12,5	62,9	0,2	24,2	13,1
4	× 2,50	5	15	76,9	0,1	21,6	7,8	61,9	0,1	25,4	8,6
4	× 2,75	5	16	71,4	0,3	27,2	1,6	56,9	0,3	31,4	2,8
4	× 3,00	5	16	71,3	0,2	27,3	1,7	57,1	0,3	31,2	2,1
8	c	4	8	46,8	0,2	0,0	30,8	38,4	0,2	0,0	35,5
8	× 1,00	5	13	51,2	0,1	-9,4	30,2	40,4	0,7	-5,2	30,2
8	× 1,25	5	15	49,0	0,1	-4,7	21,6	38,4	0,2	0,0	22,3
8	× 1,50	5	16	49,8	0,2	-6,4	21,5	38,5	0,7	-0,3	22,8
8	× 1,75	6	18	48,1	0,2	-2,8	13,8	37,4	0,2	2,6	15,2
8	× 2,00	6	24	41,8	0,4	10,7	8,5	30,6	0,3	20,3	6,4
8	× 2,25	6	28	41,3	0,3	11,8	7,5	31,2	0,5	18,8	8,2
8	× 2,50	6	31	43,7	0,7	6,6	9,8	30,8	0,6	19,8	8,1
8	× 2,75	6	32	43,8	0,5	6,4	8,4	32,4	0,4	15,6	8,7
8	× 3,00	6	32	43,8	0,2	6,4	9,3	32,3	0,3	15,9	8,1
16	c	5	16	25,4	0,2	0,0	31,4	19,8	0,2	0,0	37,2
16	× 1,00	6	24	24,6	0,4	3,1	26,5	17,3	0,3	12,6	26,3
16	× 1,25	6	31	27,4	0,4	-7,9	18,9	19,9	0,5	-0,5	19,2
16	× 1,50	6	32	26,5	0,3	-4,3	18,1	18,9	0,4	4,5	17,7
16	× 1,75	7	35	25,0	0,7	1,6	12,3	16,1	0,3	18,7	10,6
16	× 2,00	7	44	25,4	0,3	0,0	9,2	16,7	0,5	15,7	9,8
16	× 2,25	7	56	25,8	0,5	-1,6	10,1	16,4	0,6	17,2	8,2
16	× 2,50	7	62	26,1	0,4	-2,8	8,8	16,1	0,7	18,7	6,8
16	× 2,75	7	64	25,5	0,3	-0,4	8,5	16,5	0,6	16,7	7,2
16	× 3,00	8	65	26,1	0,4	-2,8	6,3	16,2	0,6	18,2	5,1

Tabla 3.1: Rendimiento de la resolución progresiva (Algoritmo 3.12) y regresiva (Algoritmo 3.14) en la aplicación de los precondicionadores construidos por el Algoritmo 2.21 para el sistema de ecuaciones que surge de la discretización, mediante una malla de 1000×1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (a).

		Árbol de tareas		Resolución Progresiva Paralela				Resolución Regresiva Paralela			
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (ms)	σ (ms)	Acel. (%)	cv (%)
1	× 1,00	1	1	345,8	0,3	0,0	0,0	284,2	0,1	0,0	0,0
2	c	2	2	182,5	0,5	0,0	7,7	147,4	0,2	0,0	7,2
2	× 1,00	3	3	162,2	0,3	11,1	1,9	134,7	0,3	8,6	2,7
2	× 1,25	3	4	150,4	0,3	17,6	0,9	125,2	0,4	15,1	0,4
2	× 1,50	3	4	150,8	0,4	17,4	1,0	124,8	0,4	15,3	0,4
2	× 1,75	3	4	151,1	0,3	17,2	1,3	124,4	0,4	15,6	0,2
2	× 2,00	4	6	157,4	0,3	13,8	9,8	127,5	0,3	13,5	7,6
2	× 2,25	4	8	141,1	0,3	22,7	0,8	115,4	0,2	21,7	0,2
2	× 2,50	4	8	140,9	1,9	22,8	0,9	115,3	1,1	21,8	0,9
2	× 2,75	4	8	140,9	0,3	22,8	0,8	115,3	0,2	21,8	0,2
2	× 3,00	4	8	140,9	0,2	22,8	0,7	115,4	0,2	21,7	0,1
4	c	3	4	95,1	0,2	0,0	24,4	80,1	0,2	0,0	27,6
4	× 1,00	4	6	84,2	0,2	11,5	20,9	70,6	0,1	11,9	20,1
4	× 1,25	4	8	81,1	1,1	14,7	18,7	65,6	0,2	18,1	18,8
4	× 1,50	4	8	81,4	1,1	14,4	18,8	65,4	0,2	18,4	18,7
4	× 1,75	4	8	81,0	1,1	14,8	19,2	65,6	0,2	18,1	19,2
4	× 2,00	5	13	79,6	0,1	16,3	10,8	63,7	0,2	20,5	9,8
4	× 2,25	5	14	74,6	2,0	21,6	5,9	60,4	1,0	24,6	6,4
4	× 2,50	5	15	82,8	2,5	12,9	13,3	65,9	1,5	17,7	13,7
4	× 2,75	5	16	76,3	0,1	19,8	6,8	59,8	0,3	25,3	6,6
4	× 3,00	5	16	76,2	0,1	19,9	6,8	60,0	0,3	25,1	6,5
8	c	4	8	46,4	0,2	0,0	23,6	38,3	0,2	0,0	28,0
8	× 1,00	5	13	43,7	0,5	5,8	23,1	35,4	0,5	7,6	23,3
8	× 1,25	5	15	47,3	2,2	-1,9	21,8	35,8	1,4	6,5	22,0
8	× 1,50	5	16	46,3	0,4	0,2	16,5	35,9	0,4	6,3	16,6
8	× 1,75	6	18	44,3	0,9	4,5	12,6	34,9	0,5	8,9	14,2
8	× 2,00	6	24	41,8	0,4	9,9	8,1	30,9	0,4	19,3	6,9
8	× 2,25	6	28	42,2	0,5	9,1	9,0	31,5	0,3	17,8	8,9
8	× 2,50	6	31	41,3	0,3	11,0	4,1	30,7	0,3	19,8	6,4
8	× 2,75	6	32	43,0	0,8	7,3	6,3	30,4	0,5	20,6	4,9
8	× 3,00	6	32	43,0	0,7	7,3	5,3	30,4	0,5	20,6	4,8
16	c	5	16	24,9	0,4	0,0	26,3	19,7	0,2	0,0	31,4
16	× 1,00	6	24	26,2	0,3	-5,2	27,4	18,7	0,4	5,1	26,7
16	× 1,25	6	31	27,0	0,3	-8,4	21,2	18,2	0,6	7,6	19,5
16	× 1,50	6	32	24,9	0,6	0,0	18,8	17,8	0,6	9,6	18,0
16	× 1,75	7	35	25,7	0,3	-3,2	13,9	18,2	0,4	7,6	14,1
16	× 2,00	7	44	26,1	0,3	-4,8	12,3	17,6	0,3	10,7	12,5
16	× 2,25	7	56	26,1	0,2	-4,8	7,7	17,0	0,5	13,7	9,1
16	× 2,50	7	62	26,1	0,3	-4,8	9,2	17,3	0,5	12,2	8,8
16	× 2,75	7	64	26,2	0,7	-5,2	5,9	16,6	0,5	15,7	7,0
16	× 3,00	8	65	26,0	0,4	-4,4	5,7	16,0	0,3	18,8	4,8

Tabla 3.2: Rendimiento de la resolución progresiva (Algoritmo 3.12) y regresiva (Algoritmo 3.14) en la aplicación de los precondicionadores construidos por el Algoritmo 2.21 para el sistema de ecuaciones que surge de la discretización, mediante una malla de 1000×1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (b).

		Árbol de tareas		Resolución Progresiva Paralela				Resolución Regresiva Paralela			
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (ms)	σ (ms)	Acel. (%)	cv (%)
1	× 1,00	1	1	346,5	0,2	0,0	0,0	284,3	0,1	0,0	0,0
2	c	2	2	182,3	0,5	0,0	7,5	147,2	0,2	0,0	7,0
2	× 1,00	3	3	162,4	0,3	10,9	1,9	135,0	0,2	8,3	3,0
2	× 1,25	3	4	152,7	0,4	16,2	2,4	126,8	0,6	13,9	2,0
2	× 1,50	3	4	152,7	0,3	16,2	2,5	126,9	0,6	13,8	2,1
2	× 1,75	3	4	152,3	0,3	16,5	2,5	127,1	0,5	13,7	2,5
2	× 2,00	4	6	145,4	0,4	20,2	1,3	119,7	0,3	18,7	1,4
2	× 2,25	4	8	143,0	0,3	21,6	2,7	118,4	0,2	19,6	3,9
2	× 2,50	4	8	142,8	2,0	21,7	2,5	118,3	1,2	19,6	3,7
2	× 2,75	4	8	143,1	0,3	21,5	2,5	118,2	0,3	19,7	3,6
2	× 3,00	4	8	143,0	2,0	21,6	2,6	118,2	1,2	19,7	3,7
4	c	3	4	82,1	0,3	0,0	8,5	68,3	0,2	0,0	9,3
4	× 1,00	4	6	76,5	0,2	6,8	4,7	62,9	0,4	7,9	5,0
4	× 1,25	4	8	72,7	0,2	11,4	5,6	60,0	0,2	12,2	6,8
4	× 1,50	4	8	72,6	0,1	11,6	5,4	60,1	0,2	12,0	6,8
4	× 1,75	4	8	72,7	0,2	11,4	5,7	60,1	0,2	12,0	6,9
4	× 2,00	5	13	75,1	0,2	8,5	11,0	61,1	0,2	10,5	11,3
4	× 2,25	5	14	77,4	0,1	5,7	7,9	62,6	0,1	8,3	8,4
4	× 2,50	5	15	74,6	0,2	9,1	6,8	59,2	0,5	13,3	6,9
4	× 2,75	5	16	71,8	0,1	12,5	2,0	57,7	0,1	15,5	2,9
4	× 3,00	5	16	71,8	0,1	12,5	2,0	57,7	0,1	15,5	2,8
8	c	4	8	46,4	0,2	0,0	18,8	38,0	0,2	0,0	21,5
8	× 1,00	5	13	44,8	0,2	3,4	17,2	34,3	0,1	9,7	17,4
8	× 1,25	5	15	41,8	0,1	9,9	15,3	33,5	0,3	11,8	16,1
8	× 1,50	5	16	41,3	0,1	11,0	6,8	32,9	0,3	13,4	9,6
8	× 1,75	6	18	42,9	0,2	7,5	11,0	32,6	0,2	14,2	10,7
8	× 2,00	6	24	47,2	1,4	-1,7	13,0	35,1	0,9	7,6	12,7
8	× 2,25	6	28	41,5	0,2	10,6	9,5	31,5	0,5	17,1	9,4
8	× 2,50	6	31	44,1	0,9	5,0	9,1	31,2	0,8	17,9	8,3
8	× 2,75	6	32	42,3	0,5	8,8	6,9	30,2	0,3	20,5	4,9
8	× 3,00	6	32	42,0	0,4	9,5	7,5	30,7	0,2	19,2	6,1
16	c	5	16	24,0	0,3	0,0	23,5	18,7	0,2	0,0	27,1
16	× 1,00	6	24	26,3	0,4	-9,6	21,9	18,0	0,5	3,7	21,8
16	× 1,25	6	31	26,0	0,4	-8,3	18,9	17,4	0,5	7,0	17,4
16	× 1,50	6	32	24,9	0,6	-3,7	15,6	17,2	0,5	8,0	14,0
16	× 1,75	7	35	24,1	0,3	-0,4	9,5	17,1	0,4	8,6	9,6
16	× 2,00	7	44	26,7	0,6	-11,2	11,4	17,0	0,5	9,1	7,9
16	× 2,25	7	56	27,5	1,0	-14,6	9,2	17,3	0,6	7,5	9,7
16	× 2,50	7	62	26,0	0,8	-8,3	7,7	16,6	0,6	11,2	6,7
16	× 2,75	7	64	26,6	0,5	-10,8	9,8	16,8	0,5	10,2	8,1
16	× 3,00	8	65	27,1	0,6	-12,9	9,5	16,4	0,6	12,3	7,9

Tabla 3.3: Rendimiento de la resolución progresiva (Algoritmo 3.12) y regresiva (Algoritmo 3.14) en la aplicación de los precondicionadores construidos por el Algoritmo 2.21 para el sistema de ecuaciones que surge de la discretización, mediante una malla de 1000×1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (c).

		Árbol de tareas		Resolución Progresiva Paralela				Resolución Regresiva Paralela			
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (ms)	σ (ms)	Acel. (%)	cv (%)
1	× 1,00	1	1	347,5	0,3	0,0	0,0	285,7	0,1	0,0	0,0
2	c	2	2	182,4	0,5	0,0	7,4	147,5	0,2	0,0	7,2
2	× 1,00	3	3	162,4	0,3	11,0	2,0	135,0	0,2	8,5	2,9
2	× 1,25	3	4	150,2	0,4	17,7	0,5	125,2	0,5	15,1	0,2
2	× 1,50	3	4	150,4	1,1	17,5	0,7	125,1	0,9	15,2	0,2
2	× 1,75	3	4	150,3	0,3	17,6	0,8	125,1	0,4	15,2	0,1
2	× 2,00	4	6	144,3	0,2	20,9	0,5	119,6	0,3	18,9	0,5
2	× 2,25	4	8	141,1	0,3	22,6	0,1	115,4	0,3	21,8	0,5
2	× 2,50	4	8	140,9	0,3	22,8	0,3	115,7	0,3	21,6	1,1
2	× 2,75	4	8	141,1	0,3	22,6	0,1	115,5	0,3	21,7	0,8
2	× 3,00	4	8	140,8	0,3	22,8	0,1	115,6	0,2	21,6	0,9
4	c	3	4	82,4	0,2	0,0	9,2	68,2	0,2	0,0	9,7
4	× 1,00	4	6	75,9	0,2	7,9	5,3	61,6	0,3	9,7	5,2
4	× 1,25	4	8	73,2	0,2	11,2	3,8	60,7	0,5	11,0	5,3
4	× 1,50	4	8	73,4	0,2	10,9	4,0	60,8	0,5	10,9	5,4
4	× 1,75	4	8	73,3	0,2	11,0	3,9	60,6	0,5	11,1	5,3
4	× 2,00	5	13	71,8	0,2	12,9	3,0	58,2	0,3	14,7	3,0
4	× 2,25	5	14	74,2	1,1	10,0	3,8	59,2	0,8	13,2	4,1
4	× 2,50	5	15	74,1	0,2	10,1	4,1	58,2	0,6	14,7	4,1
4	× 2,75	5	16	72,3	0,2	12,3	1,3	58,1	0,2	14,8	3,3
4	× 3,00	5	16	72,2	0,2	12,4	1,2	58,2	0,2	14,7	3,3
8	c	4	8	40,8	0,1	0,0	13,9	33,2	0,2	0,0	15,1
8	× 1,00	5	13	40,6	0,2	0,5	11,9	32,0	0,3	3,6	12,5
8	× 1,25	5	15	39,0	2,8	4,4	7,3	30,9	1,4	6,9	9,7
8	× 1,50	5	16	38,9	0,1	4,7	5,7	30,8	0,2	7,2	7,4
8	× 1,75	6	18	43,9	0,1	-7,6	9,1	34,3	0,1	-3,3	8,6
8	× 2,00	6	24	43,1	1,0	-5,6	6,5	32,1	1,1	3,3	5,5
8	× 2,25	6	28	40,1	0,8	1,7	3,8	30,1	0,6	9,3	4,4
8	× 2,50	6	31	44,6	0,2	-9,3	7,5	33,6	0,5	-1,2	6,6
8	× 2,75	6	32	42,9	0,3	-5,1	4,4	32,9	0,3	0,9	6,2
8	× 3,00	6	32	43,0	0,3	-5,4	4,6	32,8	0,2	1,2	6,1
16	c	5	16	24,5	0,2	0,0	20,0	19,2	1,5	0,0	23,5
16	× 1,00	6	24	23,8	0,9	2,9	17,4	17,4	0,3	9,4	18,9
16	× 1,25	6	31	25,1	0,6	-2,4	12,5	18,0	0,7	6,2	13,3
16	× 1,50	6	32	24,7	0,3	-0,8	10,4	17,2	0,3	10,4	12,1
16	× 1,75	7	35	23,3	0,5	4,9	6,7	15,7	0,4	18,2	7,3
16	× 2,00	7	44	23,9	0,5	2,4	9,2	16,8	0,4	12,5	9,8
16	× 2,25	7	56	25,4	0,4	-3,7	8,5	17,2	0,7	10,4	8,7
16	× 2,50	7	62	25,4	0,4	-3,7	6,1	17,1	0,5	10,9	6,6
16	× 2,75	7	64	25,6	0,3	-4,5	5,3	16,5	0,3	14,1	5,8
16	× 3,00	8	65	26,1	0,4	-6,5	5,6	16,8	0,4	12,5	5,4

Tabla 3.4: Rendimiento de la resolución progresiva (Algoritmo 3.12) y regresiva (Algoritmo 3.14) en la aplicación de los precondicionadores construidos por el Algoritmo 2.21 para el sistema de ecuaciones que surge de la discretización, mediante una malla de 1000×1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (d).

estos factores en las prestaciones depende de cada problema. Por ejemplo, en el caso del problema de la Tabla 3.1, la aceleración correspondiente a $p = 4/f = p \times 2{,}75$ fue del 27,2 % y el 31,4 % para la resolución progresiva y regresiva, respectivamente, mientras que en la Tabla 3.4 (ver Figura 2.20 (d)), sólo fue del 12,5 % y el 15,5 %. En el primer caso, la aceleración está causada por la combinación de ambos factores, mientras que, en el segundo, *principalmente* por el mejor aprovechamiento de la localidad por parte de las cachés de los procesadores.

Los resultados obtenidos con 8 y 16 procesos no evidencian el mismo comportamiento discutido en el párrafo anterior. En primer lugar, no siempre se obtiene una reducción del tiempo de ejecución paralelo escogiendo el valor de f en el intervalo $[p, 3p]$ (los valores de la columna "Acel. (%)" sólo son positivos en algunos casos) y, además, esta elección puede conllevar un aumento (los valores de la columna "Acel. (%)" son negativos en algunos casos). Este aumento fue, en el peor de los casos, del 14,6 %, para la resolución progresiva (ver Tabla 3.3, $p = 16/f = p \times 2{,}25$), y del 5,2 %, para la regresiva (ver Tabla 3.1, $p = 8/f = p \times 1{,}00$). En segundo lugar, no existe una tendencia clara hacia la reducción del tiempo de ejecución cuando f crece, a pesar de que se consigue una distribución más equilibrada de la carga de trabajo (los valores de la columna "*cv* (%)" generalmente disminuyen). Este resultado contradictorio puede estar causado por el sobrecoste que supone el acceso a los bloques correspondientes a los ancestros de las nuevas tareas obtenidas al aumentar f. Al dividir el cómputo en árboles de tareas con mayor número de niveles/hojas, cada proceso recupera de memoria un mayor volumen de datos (aumenta el tráfico entre el sistema de memoria y las cachés de los procesadores), ya que las nuevas tareas que se obtienen almacenan en sus vectores locales los bloques correspondientes a un mayor número de ancestros. Los resultados de las tablas revelan que este sobrecoste afecta en menor grado a la resolución progresiva paralela. Esta última observación se puede justificar teniendo en cuenta que los procesos ejecutan las tareas (acceden a sus datos) de la resolución regresiva a continuación de las tareas de la progresiva y, por tanto, el mejor aprovechamiento de la localidad temporal en las cachés de los procesadores permite ocultar en cierto grado el sobrecoste de acceso a los bloques locales asociados a los ancestros de las nuevas tareas.

El siguiente experimento cuantifica el sobrecoste que supone dividir el cómputo de la resolución progresiva y regresiva en árboles con un mayor número de niveles y/o tareas. Para ello, ejecutamos los Algoritmos 3.12 y 3.14 con un sólo proceso, y medimos el tiempo de ejecución para valores de f progresivamente más grandes (en particular, los mismos valores de las Tablas 3.1, 3.2, 3.3 y 3.4). La Figura 3.5 ilustra, para la resolución progresiva (aspas) y regresiva (círculos), el *ratio* entre el tiempo de ejecución del algoritmo paralelo ejecutado con un sólo proceso y el tiempo de ejecución del algoritmo secuencial. Así, valores superiores a la unidad en esta figura indican que el coste agregado del algoritmo paralelo es superior al del algoritmo secuencial, mientras que valores inferiores a la unidad indican lo contrario. La Figura 3.5 revela claramente los factores mencionados en la discusión de los resultados de las tablas; en primera instancia, el ratio disminuye debido al mejor aprovechamiento de la localidad y, a partir de cierto valor de f, empieza a dominar el sobrecoste que supone recuperar de memoria mayor volumen de datos (el ratio aumenta). Este ratio aumenta más rápido para la resolución progresiva que la regresiva, lo que justifica la diferencia entre las aceleraciones de ambos algoritmos en las Tablas 3.1, 3.2, 3.3 y 3.4. Es importante remarcar que los resultados de la figura se obtuvieron ejecutando los algoritmos paralelos *con un sólo procesador físico*. En el sistema multiprocesador con el que hemos obtenido los resultados, el ancho de banda agregado crece con el número de procesadores físicos y, por tanto, al ejecutar el algoritmo paralelo con varios procesadores, parte de este sobrecoste se reduce por el efecto del paralelismo.

Comparativa del rendimiento de los Algoritmos 3.13 y 3.14

Entre una ejecución y otra distinta de la terna formada por: (1) Algoritmo 2.21; (2) Algoritmo 3.19 utilizando los Algoritmos 3.12 y 3.13; y (3) Algoritmo 3.19 utilizando los Algoritmos 3.12 y 3.14, el mapeado de tareas a procesos puede cambiar y, por tanto, el rendimiento relativo de los

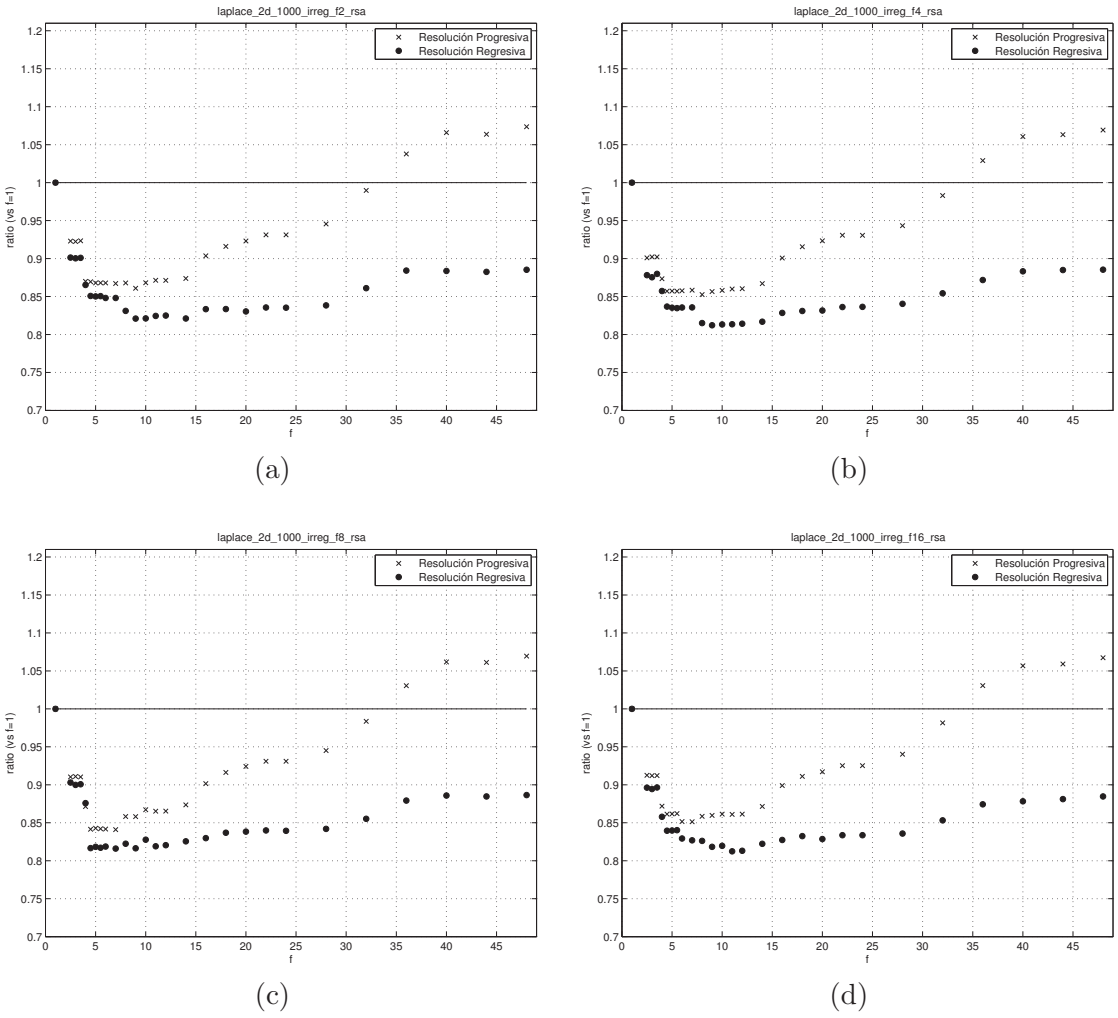


Figura 3.5: Sobrecoste que supone dividir el cómputo de la resolución progresiva (aspas) y regresiva (círculos) en árboles con mayor número de niveles y/o tareas para los cuatro problemas de la Figura 2.20 (a), (b), (c) y (d).

Algoritmos 3.13 y 3.14 puede resultar afectado. El siguiente experimento compara el rendimiento de estos algoritmos, y cuantifica, en consecuencia, el impacto que tienen sobre las prestaciones los ajustes que realiza el Algoritmo 3.14 en el mapeado inicial de tareas a procesos. Con este propósito, hemos registrado el *tiempo medio por iteración* que supone la resolución regresiva paralela en cada una de las 50 ejecuciones de (2) y (3).

Las Tablas 3.5, 3.6, 3.7 y 3.8 resumen los resultados de la comparativa para los cuatro problemas de la Figura 2.20. La parte superior de las tablas muestra, para cada combinación de p y f, el número de ejecuciones (sobre un total de 50) para las cuales *el Algoritmo 3.14 fue más del 10 % más rápido que el Algoritmo 3.13*; es decir, aquellos casos en los que, partiendo del mismo mapeado de tareas a procesos, el incremento relativo del tiempo de ejecución del Algoritmo 3.13 con respecto al del Algoritmo 3.14 es superior al 10 %. Además, entre paréntesis, proporcionamos la media de los incrementos relativos para cada una de estas ejecuciones. La parte inferior de las tablas muestra la misma información (número de ejecuciones y media de los incrementos relativos entre paréntesis) para aquellos casos en los que el Algoritmo 3.13 fue más del 10 % más rápido que el Algoritmo 3.14. Por último, se han omitido intencionadamente los resultados de los experimentos para los cuales no hubo ninguna ejecución que cumplió los criterios descritos. Por ejemplo, las filas correspondientes a $p = 2, 4, 8, 16$ procesos se han omitido de la parte inferior de la Tabla 3.7, ya que no hubo ninguna ejecución para la cual el Algoritmo 3.13

fue más del 10 % más rápido que el Algoritmo 3.14.

Los resultados de las tablas evidencian claramente el impacto *positivo* que tienen sobre las prestaciones de la resolución regresiva paralela los ajustes que realiza el Algoritmo 3.14 en el mapeado inicial de tareas a procesos. Por ejemplo, en la Tabla 3.7, el Algoritmo 3.14 fue, en las 50 ejecuciones de la combinación $p = 4/f = 4$, un 75 % más rápido (de media) que el Algoritmo 3.13 y, en 38 ejecuciones de la combinación $p = 16/f = 16$, un 26 % más rápido (de media). Además, en la mayoría de las combinaciones de p y f para las que el Algoritmo 3.14 no fue más del 10 % más rápido que el Algoritmo 3.13, el rendimiento de ambos algoritmos fue comparable (ninguno de los algoritmos fue más del 10 % más rápido que el otro). Por último, en *sólo una* de las ejecuciones de tres de las opciones consideradas ($p = 16/f = 20$, $p = 16/f = 48$ en la Tabla 3.5, y $p = 16/f = 20$ en la Tabla 3.5), el Algoritmo 3.13 fue más del 10 % más rápido que el Algoritmo 3.14. Esta última observación evidencia que los ajustes que realiza el Algoritmo 3.14 pueden tener un impacto *negativo* en las prestaciones. Como se indicó en el Apartado 3.3.3, al realizar un ajuste en el mapeado de una determinada tarea con hijos, la expectativa es que esta tarea (y sus descendientes) tengan que esperar menos tiempo en la cola del nuevo proceso, que en la del proceso que inicialmente estaba a cargo de su ejecución. Sin embargo, teniendo en cuenta la información que maneja el Algoritmo 3.14, no existen garantías de que esta expectativa se cumpla. Es necesario remarcar que los resultados de las tablas revelan que la penalización en la que se incurre en ese caso es relativamente moderada (el Algoritmo 3.13 sólo fue, en el peor de los casos, el 14 % más rápido que el Algoritmo 3.14).

Alg. 3.14 > 10 % más rápido que Alg. 3.13									
p/f	×1,00	×1,25	×1,50	×1,75	×2,00	**×2,25**	×2,50	×2,75	×3,00
8	6 (19 %)	0	0	0	2 (15 %)	14 (18 %)	0	0	0
16	21 (24 %)	3 (11 %)	14 (18 %)	16 (14 %)	15 (13 %)	**2 (11 %)**	1 (11 %)	1 (11 %)	2 (10 %)
Alg. 3.13 > 10 % más rápido que Alg. 3.14									
p/f	×1,00	×1,25	×1,50	×1,75	×2,00	**×2,25**	×2,50	×2,75	×3,00
16	0	0	0	0	0	**1 (14 %)**	0	0	1 (12 %)

Tabla 3.5: Resumen de la comparativa de los Algoritmos 3.13 y 3.14 para el problema de la Figura 2.20 (a).

Alg. 3.14 > 10 % más rápido que Alg. 3.13									
p/f	**×1,00**	×1,25	×1,50	×1,75	×2,00	×2,25	×2,50	×2,75	×3,00
4	41 (37 %)	0	0	0	1 (11 %)	0	0	0	0
8	5 (23 %)	26 (31 %)	1 (14 %)	0	4 (17 %)	0	0	0	0
16	**30 (28 %)**	3 (20 %)	3 (15 %)	1 (13 %)	9 (14 %)	5 (17 %)	0	0	0

Tabla 3.6: Resumen de la comparativa de los Algoritmos 3.13 y 3.14 para el problema de la Figura 2.20 (b).

Alg. 3.14 > 10 % más rápido que Alg. 3.13									
p/f	×1,00	×1,25	×1,50	×1,75	×2,00	×2,25	×2,50	×2,75	×3,00
4	50 (75 %)	4 (18 %)	0	1 (25 %)	5 (14 %)	0	0	0	0
8	2 (23 %)	2 (11 %)	0	3 (11 %)	3 (22 %)	3 (15 %)	0	0	0
16	38 (26 %)	8 (13 %)	5 (20 %)	7 (16 %)	11 (11 %)	3 (15 %)	0	0	0

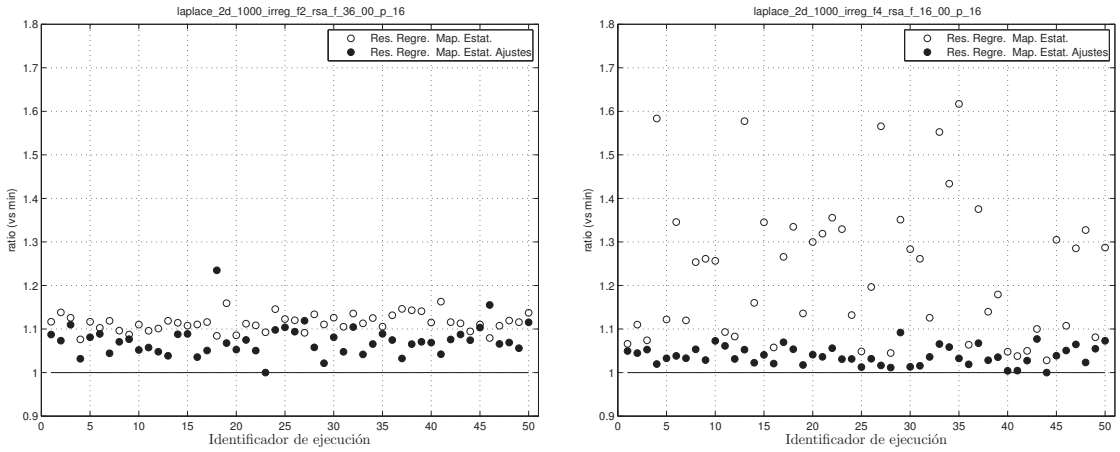
Tabla 3.7: Resumen de la comparativa de los Algoritmos 3.13 y 3.14 para el problema de la Figura 2.20 (c).

Para terminar la comparativa, analizamos con más detalle el rendimiento relativo de ambos algoritmos para las dos combinaciones de p y f marcadas en negrita en las Tablas 3.5 y 3.6. Las Figuras 3.6 (a) y (b) representan, para $p = 16/f = 36$ (Tabla 3.5) y $p = 16/f = 16$ (Tabla 3.6), respectivamente, el *ratio* entre el tiempo de ejecución de los Algoritmos 3.13 (círculos blancos) y 3.14 (círculos negros), y el tiempo de ejecución mínimo entre las 50 ejecuciones de ambos

Alg. 3.14 > 10 % más rápido que Alg. 3.13									
p/f	×1,00	×1,25	×1,50	×1,75	×2,00	×2,25	×2,50	×2,75	×3,00
4	46 (50 %)	0	0	0	0	2 (23 %)	0	0	0
8	31 (33 %)	13 (26 %)	0	0	7 (18 %)	4 (12 %)	0	0	0
16	45 (44 %)	5 (18 %)	1 (15 %)	20 (16 %)	19 (16 %)	3 (11 %)	2 (15 %)	0	0
Alg. 3.13 > 10 % más rápido que Alg. 3.14									
p/f	×1,00	×1,25	×1,50	×1,75	×2,00	×2,25	×2,50	×2,75	×3,00
16	0	0	0	0	0	1 (14 %)	0	0	0

Tabla 3.8: Resumen de la comparativa de los Algoritmos 3.13 y 3.14 para el problema de la Figura 2.20 (d).

algoritmos. Por lo tanto, todos los valores representados en la figura son superiores o iguales a la unidad y, cuánto mayor sea la magnitud del valor representado, más lento es el algoritmo correspondiente. En una ejecución dada (identificada por el valor correspondiente del eje de abscisas), el mapeado inicial de tareas a procesos coincide para ambos algoritmos y, por tanto, las diferencias observadas entre los ratios se deben principalmente a los ajustes que realiza el Algoritmo 3.14 en este mapeado (comprobar, por ejemplo, la diferencia entre los ratios de ambos algoritmos para la ejecución con identificador 15 en la Figura 3.6 (b)). Como se puede observar en la Figura 3.6 (a), el Algoritmo 3.14 fue más rápido que el Algoritmo 3.13 en todas las ejecuciones, excepto la 18, la 27 y la 46 y, en los resultados que se recogen en la Figura 3.6 (b), fue siempre más rápido. Estos resultados, y los mostrados en las Tablas 3.5, 3.6, 3.7 y 3.8, evidencian claramente el beneficio que suponen los ajustes que realiza el Algoritmo 3.14 en el mapeado inicial de tareas a procesos.



(a) (b)

Figura 3.6: Comparativa del rendimiento de los Algoritmos 3.13 (círculos blancos) y 3.14 (círculos negros) para las 50 ejecuciones de las combinaciones (a) $p = 16/f = 36$ (Tabla 3.5) y (b) $p = 16/f = 16$ (Tabla 3.6).

Rendimiento paralelo del método PCG y del producto matriz-vector disperso

Las Tablas 3.9, 3.10, 3.11 y 3.12 recogen el rendimiento paralelo del producto matriz-vector disperso y del método PCG. El vector de términos independientes del sistema se ha construido artificialmente para que la solución exacta del sistema $Ax = b$ sea el vector unidad. La solución inicial escogida fue $x^{(0)} = 0$ (ver línea 1 del Algoritmo 3.19), y el proceso iterativo se detiene cuando se cumple el criterio de convergencia descrito en [152]. Este criterio emplea una estimación del error relativo en la A-norma, $\|x - x^{(i)}\|_A / \|x - x^{(0)}\|_A$, para evaluar, en cada iteración, la precisión de la solución aproximada calculada hasta el momento. Si $\|x - x^{(i)}\|_A / \|x - x^{(0)}\|_A \lessapprox \epsilon$,

donde ϵ es un parámetro predefinido, entonces se detiene el proceso iterativo. Para todos los experimentos hemos escogido $\epsilon = 10^{-8}$, el valor por defecto en la biblioteca ILUPACK. Los resultados mostrados en la Tabla 3.9 para un combinación de p y f dada se obtuvieron de la *misma ejecución* (mismo mapeado inicial de tareas a procesos) para la que se obtuvieron los de la Tabla 3.1; la misma relación se cumple para los resultados de las Tablas 3.10, 3.11 y 3.12, y 3.2, 3.3 y 3.4, respectivamente. La columna etiquetada como "T_p (ms)" muestra, en milisegundos, el tiempo medio por iteración que cuesta el Algoritmo 3.18 y, la etiquetada como "cv (%)", el *coeficiente de variación* que se obtiene como el cociente entre la media aritmética y la desviación típica de la suma de los costes computacionales de las tareas hoja asignadas a cada proceso. Por lo tanto, un valor más pequeño del coeficiente de variación indica una distribución más homogénea de la carga computacional. La columna "T_p (s)" recoge el tiempo de ejecución del método PCG en segundos y, las etiquetadas como "Acel. (%)", cuantifican la aceleración (valores positivos) o deceleración (valores negativos) relativa en el tiempo de ejecución que se obtiene empleando valores de f en el intervalo $[p, 3p]$, con respecto al que se obtiene con $f = c$. Las columnas etiquetadas como "σ (ms)" y "σ (s)" contienen la desviación estándar de la muestra formada por los tiempos de ejecución del producto matriz-vector disperso y del método PCG paralelo en cada una de las 50 ejecuciones de (3). Este indicador mide la dispersión que existe entre los tiempos de ejecución en función del mapeado inicial de tareas a procesos. Por último, las columnas etiquetadas como "#iter." y "$\|x - x^{(i)}\|_2 / \|x\|_2$" muestran, respectivamente, el número de iteraciones requeridas por el Algoritmo 3.19 para converger a la solución de (3.79), y el *error relativo* correspondiente a la solución aproximada, $x^{(i)}$, calculada por el método PCG (ver línea 23 del Algoritmo 3.19). Este último indicador, que se ha podido calcular ya que la solución exacta es conocida, mide, a posteriori, la *precisión* de la solución calculada por el método iterativo de resolución.

Antes de comenzar el análisis detallado de los resultados, es importante mencionar que, a diferencia de la resolución progresiva y regresiva, el valor de la columna "σ (ms)" sí fue, para el producto matriz-vector disperso y algunas combinaciones de p y f mostradas en las Tablas 3.9, 3.10, 3.11 y 3.12, relativamente alto en comparación con el valor de "T_p (ms)". Por ejemplo, σ (ms) fue, en la Tabla 3.10, $p = 4/f = 5$, de 2,71 ms. Por tanto, el mapeado inicial de tareas a procesos puede tener un impacto significativo sobre la dispersión de la muestra formada por los tiempos de ejecución del producto matriz-vector disperso en cada una de las 50 ejecuciones de (3). Esta variabilidad está causada por el diferente grado de homogeneidad/heterogeneidad de la distribución de la carga en el producto matriz-vector disperso paralelo al emplear un mapeado inicial de tareas a procesos diferente.

Para comenzar la discusión, nos centramos en el rendimiento del producto matriz-vector disperso. Los resultados de las tablas revelan que, generalmente, el tiempo de ejecución *aumenta* escogiendo valores de f en el intervalo $[p, 3p]$ (los valores de la columna "Acel. (%)" son negativos en la mayoría de los casos). Este aumento fue, en el peor de los casos, del 52,4 % (ver Tabla 3.9, $p = 8/f = p \times 1{,}75$). Además, no existe una tendencia uniforme del tiempo de ejecución en función del valor de f. Estos resultados se pueden justificar teniendo en cuenta los siguientes factores/aspectos:

- Al dividir el cómputo en árboles con *el mismo número de hojas que procesos* se obtiene una distribución homogénea de la carga computacional (ver valores de la columna "cv (%)" para $f = c$ en las tablas). El número de operaciones que se realizan por cada nodo (fila) del grafo de adyacencia (de la matriz de coeficientes) es constante y, por lo tanto, dado que la fase inicial de particionado tiene por objeto equilibrar el número de nodos de los subgrafos independientes, cada proceso realiza aproximadamente el mismo número de operaciones. Este argumento también justifica por qué es moderada la reducción del tiempo de ejecución que se obtiene en algunos casos al escoger valores de f en el intervalo $[p, 3p]$ (ver, por ejemplo, los valores de la columna "Acel. (%)" para $p = 2$ en la Tabla 3.12).

- Al dividir el cómputo en árboles con un número de hojas mayor que el de procesos se

		Árbol de tareas		Producto Paralelo Matriz-Vector				Método PCG paralelo				
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (s)	σ (s)	Acel. (%)	#iter.	$\frac{\|x-x^{(i)}\|_2}{\|x\|_2}$
1	× 1,00	1	1	60,2	0,07	0,0	0,0	80,8	0,04	0,0	115	2,2e-08
2	c	2	2	32,1	0,04	0,0	6,0	51,3	0,08	0,0	116	1,7e-08
2	× 1,00	3	3	43,2	0,06	-34,6	45,4	51,4	0,05	-0,2	118	1,9e-08
2	× 1,25	3	4	30,2	0,05	5,9	0,9	38,2	0,08	25,5	119	1,5e-08
2	× 1,50	3	4	30,6	0,08	4,7	0,3	38,2	0,08	25,5	119	1,5e-08
2	× 1,75	3	4	30,2	0,03	5,9	0,9	38,1	0,08	25,7	119	1,5e-08
2	× 2,00	4	6	30,3	0,06	5,6	1,0	36,6	0,07	28,7	120	2,1e-08
2	× 2,25	4	8	31,1	0,06	3,1	4,3	36,6	0,08	28,7	120	1,7e-08
2	× 2,50	4	8	31,0	0,04	3,4	4,5	36,7	0,05	28,5	120	1,7e-08
2	× 2,75	4	8	31,1	0,04	3,1	4,4	36,7	0,05	28,5	120	1,7e-08
2	× 3,00	4	8	31,1	0,05	3,1	4,5	36,7	0,06	28,5	120	1,7e-08
4	c	3	4	15,8	0,04	0,0	6,1	23,8	0,05	0,0	119	1,5e-08
4	× 1,00	4	6	21,7	0,03	-37,3	45,7	23,7	0,06	0,4	120	2,1e-08
4	× 1,25	4	8	22,8	0,05	-44,3	35,5	24,4	0,06	-2,5	120	1,7e-08
4	× 1,50	4	8	22,8	0,03	-44,3	35,7	24,4	0,07	-2,5	120	1,7e-08
4	× 1,75	4	8	22,8	0,03	-44,3	35,6	24,3	0,07	-2,1	120	1,7e-08
4	× 2,00	5	13	17,6	0,05	-11,4	11,3	19,5	0,04	18,1	121	1,7e-08
4	× 2,25	5	14	17,6	0,10	-11,4	17,0	19,1	0,03	19,7	117	2,6e-08
4	× 2,50	5	15	17,0	0,02	-7,6	13,8	18,6	0,02	21,8	117	2,6e-08
4	× 2,75	5	16	15,6	0,02	1,3	4,8	17,3	0,05	27,3	117	2,3e-08
4	× 3,00	5	16	15,6	0,02	1,3	4,8	17,2	0,05	27,7	117	2,3e-08
8	c	4	8	8,2	0,03	0,0	10,3	11,4	0,03	0,0	120	1,7e-08
8	× 1,00	5	13	12,0	0,04	-46,3	38,9	12,8	0,08	-12,3	121	1,7e-08
8	× 1,25	5	15	11,3	0,02	-37,8	30,5	11,8	0,02	-3,5	117	2,6e-08
8	× 1,50	5	16	11,8	0,03	-43,9	34,6	12,0	0,09	-5,3	117	2,3e-08
8	× 1,75	6	18	12,5	0,03	-52,4	25,6	12,4	0,03	-8,8	124	1,6e-08
8	× 2,00	6	24	9,9	0,02	-20,7	20,3	10,4	0,08	8,8	124	1,8e-08
8	× 2,25	6	28	9,2	0,03	-12,2	16,5	10,4	0,08	8,8	125	1,5e-08
8	× 2,50	6	31	10,0	0,41	-22,0	20,0	10,8	0,20	5,3	125	1,9e-08
8	× 2,75	6	32	11,4	0,38	-39,0	22,6	11,3	0,16	0,9	125	2,1e-08
8	× 3,00	6	32	11,4	0,03	-39,0	22,6	11,2	0,05	1,8	125	2,1e-08
16	c	5	16	4,7	0,03	0,0	12,5	6,0	0,03	0,0	117	2,3e-08
16	× 1,00	6	24	5,4	0,03	-14,9	26,6	6,1	0,07	-1,7	124	1,8e-08
16	× 1,25	6	31	6,3	0,05	-34,0	27,6	6,9	0,09	-15,0	125	1,9e-08
16	× 1,50	6	32	6,0	0,05	-27,7	28,5	6,7	0,08	-11,7	125	2,1e-08
16	× 1,75	7	35	5,9	0,05	-25,5	19,0	6,1	0,12	-1,7	125	1,9e-08
16	× 2,00	7	44	5,5	0,09	-17,0	17,5	6,0	0,10	0,0	121	2,2e-08
16	× 2,25	7	56	5,1	0,09	-8,5	16,5	6,2	0,10	-3,3	124	2,6e-08
16	× 2,50	7	62	5,4	0,08	-14,9	15,5	6,2	0,10	-3,3	123	2,1e-08
16	× 2,75	7	64	5,3	0,04	-12,8	17,9	6,2	0,09	-3,3	123	2,1e-08
16	× 3,00	8	65	5,4	0,04	-14,9	15,4	6,2	0,10	-3,3	123	2,1e-08

Tabla 3.9: Rendimiento paralelo del producto matriz-vector disperso (Algoritmo 3.18) y del método PCG (Algoritmo 3.19) en la resolución del sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (a).

		Árbol de tareas		Producto Paralelo Matriz-Vector				Método PCG paralelo				
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (s)	σ (s)	Acel. (%)	#iter.	$\frac{\|x-x^{(i)}\|_2}{\|x\|_2}$
1	× 1,00	1	1	60,2	0,05	0,0	0,0	85,4	0,04	0,0	121	7,5e-09
2	c	2	2	32,3	0,05	0,0	6,1	46,1	0,08	0,0	122	2,5e-08
2	× 1,00	3	3	31,4	0,06	2,8	5,3	42,7	0,06	7,4	124	1,8e-08
2	× 1,25	3	4	30,2	0,05	6,5	1,1	37,9	0,05	17,8	118	3,2e-08
2	× 1,50	3	4	30,2	0,04	6,5	0,9	37,9	0,05	17,8	118	3,2e-08
2	× 1,75	3	4	30,2	0,05	6,5	1,1	37,9	0,05	17,8	118	3,2e-08
2	× 2,00	4	6	37,7	0,04	-16,7	26,2	41,3	0,06	10,4	122	1,5e-08
2	× 2,25	4	8	30,2	0,04	6,5	1,2	37,2	0,06	19,3	123	1,9e-08
2	× 2,50	4	8	30,1	0,98	6,8	1,3	37,2	0,48	19,3	123	1,9e-08
2	× 2,75	4	8	30,1	0,05	6,8	1,2	37,2	0,05	19,3	123	1,9e-08
2	× 3,00	4	8	30,2	0,04	6,5	1,3	37,2	0,04	19,3	123	1,9e-08
4	c	3	4	15,9	0,07	0,0	6,4	22,9	0,05	0,0	118	3,2e-08
4	× 1,00	4	6	21,0	0,03	-32,1	31,8	22,0	0,04	3,9	122	1,5e-08
4	× 1,25	4	8	21,1	2,71	-32,7	31,5	21,3	0,61	7,0	123	1,9e-08
4	× 1,50	4	8	21,1	2,58	-32,7	31,6	21,3	0,58	7,0	123	1,9e-08
4	× 1,75	4	8	21,1	2,78	-32,7	31,6	21,3	0,61	7,0	123	1,9e-08
4	× 2,00	5	13	20,8	0,02	-30,8	25,4	20,8	0,02	9,2	123	1,6e-08
4	× 2,25	5	14	17,2	0,77	-8,2	13,7	19,2	0,50	16,2	123	1,6e-08
4	× 2,50	5	15	19,1	0,85	-20,1	20,7	20,8	0,62	9,2	121	3,8e-08
4	× 2,75	5	16	18,2	0,02	-14,5	16,3	18,7	0,03	18,3	118	3,9e-08
4	× 3,00	5	16	18,2	0,02	-14,5	16,3	18,7	0,03	18,3	118	3,9e-08
8	c	4	8	8,2	0,04	0,0	10,4	11,6	0,04	0,0	123	1,9e-08
8	× 1,00	5	13	10,9	0,05	-32,9	30,9	11,3	0,11	2,6	123	1,6e-08
8	× 1,25	5	15	11,5	0,20	-40,2	32,3	11,7	0,42	-0,9	121	3,8e-08
8	× 1,50	5	16	11,3	0,03	-37,8	25,6	11,3	0,09	2,6	118	3,9e-08
8	× 1,75	6	18	10,7	0,02	-30,5	24,1	10,9	0,15	6,0	119	4,0e-08
8	× 2,00	6	24	10,2	0,07	-24,4	20,8	10,3	0,08	11,2	121	3,2e-08
8	× 2,25	6	28	10,3	0,33	-25,6	21,4	10,9	0,13	6,0	127	2,3e-08
8	× 2,50	6	31	9,8	0,03	-19,5	17,1	10,7	0,08	7,8	128	2,0e-08
8	× 2,75	6	32	9,7	0,06	-18,3	12,6	10,9	0,12	6,0	128	1,9e-08
8	× 3,00	6	32	9,7	0,04	-18,3	12,1	10,9	0,10	6,0	128	1,9e-08
16	c	5	16	4,6	0,02	0,0	12,4	6,0	0,05	0,0	118	3,9e-08
16	× 1,00	6	24	5,8	0,04	-26,1	30,2	6,4	0,07	-6,7	121	3,2e-08
16	× 1,25	6	31	6,1	0,05	-32,6	26,9	6,8	0,09	-13,3	128	2,0e-08
16	× 1,50	6	32	5,9	0,03	-28,3	27,2	6,5	0,15	-8,3	128	1,9e-08
16	× 1,75	7	35	5,9	0,07	-28,3	26,2	6,7	0,08	-11,7	128	2,0e-08
16	× 2,00	7	44	5,8	0,10	-26,1	23,1	6,6	0,06	-10,0	128	2,3e-08
16	× 2,25	7	56	5,7	0,12	-23,9	17,0	6,3	0,08	-5,0	123	4,1e-08
16	× 2,50	7	62	5,3	0,15	-15,2	17,1	6,6	0,10	-10,0	130	2,2e-08
16	× 2,75	7	64	5,4	0,20	-17,4	14,8	6,5	0,16	-8,3	130	2,2e-08
16	× 3,00	8	65	5,3	0,03	-15,2	12,0	6,4	0,07	-6,7	130	2,2e-08

Tabla 3.10: Rendimiento paralelo del producto matriz-vector disperso (Algoritmo 3.18) y del método PCG (Algoritmo 3.19) en la resolución del sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (b),

		Árbol de tareas		Producto Paralelo Matriz-Vector				Método PCG paralelo				
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel, (%)	cv (%)	T_p (s)	σ (s)	Acel, (%)	#iter,	$\frac{\|x-x^{(i)}\|_2}{\|x\|_2}$
1	× 1,00	1	1	60,2	0,02	0,0	0,0	89,0	0,02	0,0	126	2,1e-08
2	c	2	2	32,2	0,05	0,0	6,1	49,1	0,09	0,0	130	1,5e-08
2	× 1,00	3	3	31,4	0,05	2,5	5,3	44,2	0,05	10,0	128	1,6e-08
2	× 1,25	3	4	30,2	0,03	6,2	1,0	41,7	0,10	15,1	128	1,4e-08
2	× 1,50	3	4	30,1	0,05	6,5	1,1	41,6	0,09	15,3	128	1,4e-08
2	× 1,75	3	4	30,3	0,07	5,9	1,1	41,7	0,08	15,1	128	1,4e-08
2	× 2,00	4	6	30,1	0,05	6,5	0,3	40,7	0,09	17,1	131	2,1e-08
2	× 2,25	4	8	31,2	0,05	3,1	4,5	40,6	0,16	17,3	131	1,4e-08
2	× 2,50	4	8	31,1	0,80	3,4	4,6	40,6	0,51	17,3	131	1,4e-08
2	× 2,75	4	8	31,1	0,06	3,4	4,5	40,5	0,16	17,5	131	1,4e-08
2	× 3,00	4	8	31,1	0,81	3,4	4,5	40,8	0,51	16,9	131	1,4e-08
4	c	3	4	15,8	0,05	0,0	6,1	21,7	0,05	0,0	128	1,4e-08
4	× 1,00	4	6	15,9	0,03	-0,6	5,9	20,8	0,06	4,1	131	2,1e-08
4	× 1,25	4	8	15,5	0,03	1,9	4,0	19,8	0,04	8,8	131	1,4e-08
4	× 1,50	4	8	15,5	0,06	1,9	4,0	19,8	0,04	8,8	131	1,4e-08
4	× 1,75	4	8	15,5	0,03	1,9	4,1	19,8	0,04	8,8	131	1,4e-08
4	× 2,00	5	13	17,6	0,04	-11,4	18,5	20,4	0,04	6,0	129	3,9e-08
4	× 2,25	5	14	17,8	0,07	-12,7	15,7	21,5	0,02	0,9	133	2,5e-08
4	× 2,50	5	15	17,1	0,22	-8,2	12,3	20,6	0,10	5,1	133	2,6e-08
4	× 2,75	5	16	16,0	0,02	-1,3	5,2	19,8	0,02	8,8	133	2,5e-08
4	× 3,00	5	16	15,9	0,02	-0,6	5,2	19,8	0,02	8,8	133	2,5e-08
8	c	4	8	8,2	0,04	0,0	10,4	12,3	0,03	0,0	131	1,4e-08
8	× 1,00	5	13	9,6	0,05	-17,1	21,2	11,7	0,03	4,9	129	3,9e-08
8	× 1,25	5	15	9,1	0,04	-11,0	19,0	11,4	0,04	7,3	133	2,6e-08
8	× 1,50	5	16	8,1	0,04	1,2	5,9	11,2	0,05	8,9	133	2,5e-08
8	× 1,75	6	18	10,8	0,04	-31,7	21,6	11,9	0,04	3,3	134	2,1e-08
8	× 2,00	6	24	11,6	0,37	-41,5	26,2	13,0	0,36	-5,7	135	1,8e-08
8	× 2,25	6	28	9,1	0,08	-11,0	17,7	11,3	0,08	8,1	134	2,0e-08
8	× 2,50	6	31	10,3	0,18	-25,6	20,8	11,9	0,21	3,3	136	1,9e-08
8	× 2,75	6	32	10,3	0,08	-25,6	17,3	11,6	0,09	5,7	136	1,9e-08
8	× 3,00	6	32	10,2	0,06	-24,4	17,5	11,6	0,07	5,7	136	1,9e-08
16	c	5	16	4,6	0,02	0,0	12,4	6,5	0,05	0,0	133	2,5e-08
16	× 1,00	6	24	6,6	0,03	-43,5	29,6	7,2	0,09	-10,8	135	1,8e-08
16	× 1,25	6	31	5,8	0,02	-26,1	26,0	7,0	0,09	-7,7	136	1,9e-08
16	× 1,50	6	32	5,8	0,03	-26,1	21,9	6,8	0,12	-4,6	136	1,9e-08
16	× 1,75	7	35	5,7	0,04	-23,9	16,8	6,6	0,07	-1,5	136	1,9e-08
16	× 2,00	7	44	5,9	0,12	-28,3	23,0	7,1	0,14	-9,2	136	2,2e-08
16	× 2,25	7	56	5,9	0,37	-28,3	20,3	7,2	0,25	-10,8	137	2,4e-08
16	× 2,50	7	62	5,5	0,07	-19,6	13,4	7,0	0,15	-7,7	139	1,9e-08
16	× 2,75	7	64	5,4	0,17	-17,4	19,7	7,1	0,12	-9,2	139	1,9e-08
16	× 3,00	8	65	6,1	0,24	-32,6	20,0	7,2	0,16	-10,8	139	1,9e-08

Tabla 3.11: Rendimiento paralelo del producto matriz-vector disperso (Algoritmo 3.18) y del método PCG (Algoritmo 3.19) en la resolución del sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (c).

		Árbol de tareas		Producto Paralelo Matriz-Vector				Método PCG paralelo				
p	f	#niveles	#hojas	T_p (ms)	σ (ms)	Acel. (%)	cv (%)	T_p (s)	σ (s)	Acel. (%)	#iter.	$\frac{\|x-x^{(i)}\|_2}{\|x\|_2}$
1	× 1,00	1	1	60,3	0,01	0,0	0,0	94,3	0,04	0,0	133	1,3e-08
2	c	2	2	32,1	0,06	0,0	6,2	51,0	0,09	0,0	135	1,2e-08
2	× 1,00	3	3	31,5	0,04	1,9	5,4	46,6	0,05	8,6	135	2,1e-08
2	× 1,25	3	4	30,2	0,04	5,9	1,1	44,0	0,08	13,7	137	1,9e-08
2	× 1,50	3	4	30,2	0,04	5,9	1,1	44,1	0,26	13,5	137	1,9e-08
2	× 1,75	3	4	30,3	0,06	5,6	0,9	44,1	0,07	13,5	137	1,9e-08
2	× 2,00	4	6	30,3	0,04	5,6	1,1	42,5	0,07	16,7	137	1,9e-08
2	× 2,25	4	8	29,9	0,04	6,9	0,2	41,7	0,07	18,2	138	1,9e-08
2	× 2,50	4	8	29,9	0,04	6,9	0,2	41,8	0,07	18,0	138	1,9e-08
2	× 2,75	4	8	29,9	0,04	6,9	0,3	41,7	0,06	18,2	138	1,9e-08
2	× 3,00	4	8	29,9	0,04	6,9	0,2	41,7	0,06	18,2	138	1,9e-08
4	c	3	4	15,8	0,04	0,0	6,3	23,2	0,05	0,0	137	1,9e-08
4	× 1,00	4	6	15,8	0,03	0,0	6,1	21,5	0,05	7,3	137	1,9e-08
4	× 1,25	4	8	15,5	0,03	1,9	4,8	21,1	0,09	9,1	138	1,9e-08
4	× 1,50	4	8	15,5	0,03	1,9	4,8	21,1	0,08	9,1	138	1,9e-08
4	× 1,75	4	8	15,5	0,03	1,9	4,9	21,1	0,08	9,1	138	1,9e-08
4	× 2,00	5	13	16,5	0,03	-4,4	7,0	20,9	0,05	9,9	139	1,3e-08
4	× 2,25	5	14	16,4	0,60	-3,8	8,5	21,5	0,36	7,3	140	9,9e-09
4	× 2,50	5	15	16,0	0,18	-1,3	6,8	21,2	0,08	8,6	140	1,5e-08
4	× 2,75	5	16	15,3	0,02	3,2	2,6	20,8	0,03	10,3	140	1,7e-08
4	× 3,00	5	16	15,2	0,03	3,8	2,4	20,8	0,03	10,3	140	1,7e-08
8	c	4	8	8,2	0,03	0,0	10,4	11,6	0,03	0,0	138	1,9e-08
8	× 1,00	5	13	8,4	0,04	-2,4	10,8	11,5	0,06	0,9	139	1,3e-08
8	× 1,25	5	15	8,3	1,28	-1,2	8,3	11,2	0,79	3,4	140	1,5e-08
8	× 1,50	5	16	8,4	0,03	-2,4	6,8	11,2	0,05	3,4	140	1,7e-08
8	× 1,75	6	18	9,7	0,14	-18,3	16,4	12,7	0,04	-9,5	141	1,7e-08
8	× 2,00	6	24	10,1	0,19	-23,2	14,8	12,4	0,29	-6,9	142	8,9e-09
8	× 2,25	6	28	8,8	0,54	-7,3	10,9	11,5	0,28	0,9	142	1,9e-08
8	× 2,50	6	31	9,8	0,03	-19,5	13,4	12,9	0,08	-11,2	143	2,2e-08
8	× 2,75	6	32	9,5	0,10	-15,9	9,9	12,6	0,07	-8,6	144	2,4e-08
8	× 3,00	6	32	9,5	0,10	-15,9	10,0	12,6	0,05	-8,6	144	2,4e-08
16	c	5	16	4,7	0,03	0,0	12,3	7,0	1,43	0,0	140	1,7e-08
16	× 1,00	6	24	4,7	0,55	0,0	12,6	6,8	0,22	2,9	142	8,9e-09
16	× 1,25	6	31	6,0	0,05	-27,7	18,8	7,3	0,18	-4,3	143	2,2e-08
16	× 1,50	6	32	5,8	0,05	-23,4	16,7	7,1	0,06	-1,4	144	2,4e-08
16	× 1,75	7	35	5,3	0,06	-12,8	14,9	6,7	0,11	4,3	144	2,7e-08
16	× 2,00	7	44	4,7	0,31	0,0	13,0	6,8	0,15	2,9	145	1,9e-08
16	× 2,25	7	56	4,8	0,07	-2,1	12,7	7,1	0,12	-1,4	144	1,6e-08
16	× 2,50	7	62	5,5	0,10	-17,0	14,7	7,3	0,11	-4,3	146	1,7e-08
16	× 2,75	7	64	5,3	0,09	-12,8	11,6	7,2	0,07	-2,9	146	1,6e-08
16	× 3,00	8	65	5,4	0,22	-14,9	13,5	7,4	0,11	-5,7	146	1,8e-08

Tabla 3.12: Rendimiento paralelo del producto matriz-vector disperso (Algoritmo 3.18) y del método PCG (Algoritmo 3.19) en la resolución del sistema de ecuaciones que surge de la discretización, mediante una malla de 1000 × 1000 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (d).

obtiene generalmente una distribución *más heterogénea* de la carga computacional (ver, por ejemplo, los valores de la columna "cv (%)" para $p = 8$ y $f \in [p, 3p]$ en la Tabla 3.9) y, además, el grado de heterogeneidad de esta distribución depende en gran medida del mapeado inicial de tareas a procesos (no existe una tendencia uniforme de los valores de la columna "cv (%)" en función del valor de f). Como se expuso en el Apartado 3.4.3, el mapeado que resulta del proceso de construcción del precondicionador se mantiene para el producto matriz-vector disperso (con el objeto reducir la comunicación entre los procesadores y favorecer un mejor aprovechamiento de la localidad). El número de operaciones que realiza el Algoritmo 2.21 por cada subgrafo independiente depende de la distribución de los coeficientes de la EDP en el dominio computacional y *no es proporcional a su número de nodos*. Por lo tanto, no hay garantías de que la asignación de hojas a procesos que resulta de este algoritmo logre distribuir homogéneamente la carga computacional del producto matriz-vector disperso.

- Al dividir el cómputo en árboles de tareas con mayor número de niveles/hojas, se produce un aumento del número de operaciones y del volumen de datos que cada proceso debe recuperar de memoria, ya que las nuevas tareas hoja que se obtienen almacenan en sus *vectores y matrices* locales los bloques correspondientes a un mayor número de ancestros. Para cuantificar este sobrecoste, hemos ejecutado el Algoritmo 3.18 con un sólo proceso, y hemos medido el tiempo de ejecución para valores de f progresivamente más grandes. La Figura 3.7 representa, con círculos, el *ratio* entre el tiempo de ejecución del algoritmo paralelo ejecutado con un sólo proceso y el tiempo de ejecución del algoritmo secuencial. Así, valores superiores a la unidad en esta figura indican que el coste agregado del algoritmo paralelo es superior al del algoritmo secuencial, mientras que valores inferiores a la unidad indican lo contrario. La Figura 3.7 revela que el ratio aumenta con f y, por tanto, pone de manifiesto el efecto de este factor de sobrecoste sobre las prestaciones. El valor de este ratio se puede estimar cuantificando el número de operaciones adicionales que realiza el algoritmo paralelo en relación al algoritmo secuencial; la Figura 3.7 representa, con aspas, el *ratio* del número de operaciones que realizan ambos algoritmos. Como se puede observar en la figura, esta estimación reproduce fielmente la tendencia observada para las mediciones (círculos), confirmando como causa principal de este sobrecoste el número de operaciones (y accesos a memoria) adicionales que realiza el algoritmo paralelo en comparación con el algoritmo secuencial.

El rendimiento del método PCG paralelo se puede justificar considerando las prestaciones de la aplicación del precondicionador (Tablas 3.1, 3.2, 3.3 y 3.4) y las del producto matriz-vector disperso (Tablas 3.9, 3.10, 3.11 y 3.12), y el número de iteraciones que requiere el método para converger a la solución del sistema en cada caso (valores de la columna "#iter."). Por tanto, no estimamos necesario explicar detalladamente el rendimiento del método PCG paralelo. Sin embargo, cabe remarcar que el número de iteraciones y la precisión de la solución calculada por el método PCG paralelo fueron, *en todos los casos*, comparables a las del método secuencial. Por ejemplo, en la Tabla 3.9, el número de iteraciones del método paralelo varía entre 116 y 125, frente a las 115 del método secuencial, y el error relativo varía entre $1{,}5\times10^{-8}$ y $2{,}6\times10^{-8}$, frente a $2{,}2\times10^{-8}$. Estos resultados evidencian que el enfoque de paralelización escogido para la construcción del precondicionador (ver Sección 2.3) preserva la semántica inherente a las técnicas de precondicionado de la biblioteca ILUPACK; la Sección 4.4 proporciona evidencias adicionales referentes a esta afirmación.

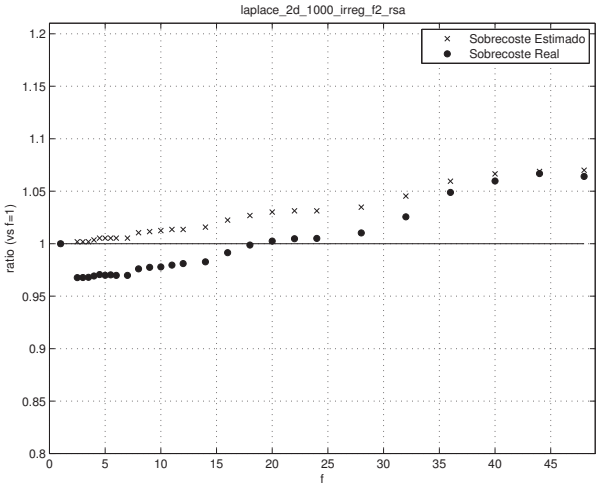


Figura 3.7: Sobrecoste estimado (aspas) y real (círculos) que supone dividir el cómputo del producto matriz-vector disperso en árboles con mayor número de niveles y/o tareas para (cualquiera de) los problemas de la Figura 2.20.

Capítulo 4

Resultados computacionales

En este capítulo presentamos y analizamos los resultados obtenidos al aplicar nuestros algoritmos paralelos para resolver los sistemas de ecuaciones lineales dispersos y de gran dimensión que aparecen ligados a una EDP elíptica tridimensional. Asimismo, para considerar problemas más irregulares, que surjan en diversas áreas de aplicación, también aplicamos nuestro enfoque para resolver en paralelo una miscelánea de sistemas de ecuaciones lineales de gran dimensión obtenidos de la colección de matrices dispersas de la Universidad de Florida [55]. El capítulo proporciona evidencias experimentales del grado de eficiencia y robustez del enfoque de paralelización, empleando para ello diversas métricas como el tiempo de resolución, la aceleración paralela, la velocidad de convergencia (número de iteraciones), la calidad (precisión) de la solución obtenida, la memoria consumida por el método para construir el precondicionador, y el tamaño ocupado por el precondicionador en memoria (número de elementos no nulos), con distintos tamaños de matrices y número de procesadores. También se proporcionan resultados experimentales adicionales que confirman la similitud entre la jerarquía multinivel construida por el método secuencial de ILUPACK y la construida por nuestro algoritmo paralelo. Además, consideramos la aplicación de aproximaciones paralelas para la etapa inicial de particionado, evaluando comparativamente su consumo de memoria, eficiencia paralela, y el impacto que tiene el particionado calculado por estas soluciones sobre las prestaciones de las etapas numéricas de nuestro resolutor. Por último, evaluamos nuestro enfoque de paralelización para resolver una miscelánea de sistemas obtenidos de la colección de matrices dispersas de la Universidad de Florida. Esta evaluación confirma que el enfoque también obtiene aceleraciones paralelas notables para sistemas con estructura irregular provenientes de problemas de aplicación muy diversos.

La estructura del capítulo es la siguiente. En la Sección 4.1 describimos el problema de aplicación escogido, y justificamos su elección. Posteriormente, en la Sección 4.2 describimos la configuración de cada una de las etapas que constituyen nuestro enfoque para el aprovechamiento del paralelismo multihebra, y la del entorno de ejecución paralelo. En la Sección 4.3 evaluamos experimentalmente los algoritmos paralelos desarrollados para el cálculo del precondicionador LDL^T multinivel y la resolución iterativa del sistema y, en la Sección 4.4, comparamos la jerarquía multinivel de los precondicionadores secuencial y paralelo. Esta comparación refuerza nuestra hipótesis de que nuestro enfoque para la extracción de paralelismo de tareas mantiene la semántica de las técnicas de precondicionado de la biblioteca ILUPACK. En la Sección 4.5, evaluamos comparativamente el rendimiento de ParMETIS y PT-SCOTCH, dos soluciones paralelas para el aprovechamiento de paralelismo durante la etapa inicial de particionado. Por último, en la Sección 4.6 evaluamos nuestro enfoque de paralelización con una miscelánea de matrices de la Universidad de Florida.

4.1. Descripción del problema de aplicación

Para evaluar la paralelización propuesta en esta tesis consideramos la resolución numérica de la EDP tridimensional

$$-\nabla \cdot K(x,y,z)\ \nabla u(x,y,z) = f, \tag{4.1}$$

en el dominio computacional, Ω, mostrado en la Figura 4.1, donde los coeficientes $K(x,y,z) \in \mathbb{R}^{3\times 3}$ son positivos y aleatorios en Ω. Para la discretización de esta EDP se han empleado elementos finitos lineales y la herramienta de generación automática y refinamiento de mallas NETGEN [147]. Partiendo de la geometría CSG (*Constructive Solid Geometry*) de Ω, NETGEN se utilizó, en primer lugar, para generar automáticamente una malla inicial en base a uno de los cinco niveles de mallado predefinidos en la herramienta: *very coarse, coarse, moderate, fine y very fine*. La Figura 4.1 ilustra la malla inicial correspondiente a uno de estos niveles de mallado. Posteriormente, esta malla se refinó una, dos o tres veces. El tamaño y número de elementos no nulos de los sistemas SDP resultantes dependen de la malla inicial, y del número de refinamientos posteriores de ésta. La Tabla 4.1 presenta, para cada una de las 12 matrices de prueba que hemos seleccionado para nuestro estudio experimental, su identificador, su código, el nivel de mallado de la malla inicial, el número de refinamientos posteriores de esta malla, su número de filas/columnas y su número medio de entradas no nulas por cada fila. El vector de términos independientes se ha construido artificialmente para que la solución exacta del sistema $Ax = b$ sea el vector unidad.

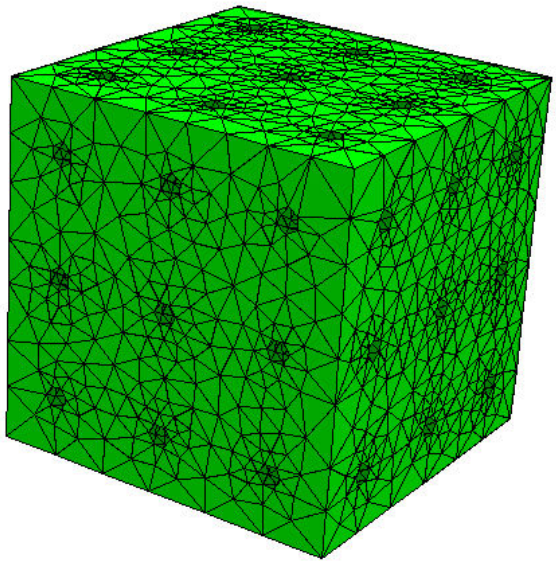


Figura 4.1: Dominio computacional con huecos cilíndricos en su interior y malla inicial generada automáticamente por NETGEN.

Las razones que justifican la elección de este problema de aplicación para nuestro estudio experimental son las siguientes:

- La geometría subyacente del problema es *tridimensional* y, como consecuencia, los métodos directos de resolución de sistemas dispersos escalan pobremente con el tamaño del sistema en términos de número de operaciones y consumo de memoria (debido al llenado masivo que se produce en el proceso de factorización).
- La EDP se resuelve en un dominio computacional que presenta huecos cilíndricos en su interior y, además, la malla de elementos finitos presenta cierto grado de irregularidad causado por el refinamiento en las zonas cercanas a las fronteras de Ω. Esta irregularidad

Id.	Código	Nivel de mallado	# refs.	n	$nnz(A)$	$nnz(A)/n$
1	VC	*very coarse*	0	1.709	16.669	9,75
2	C	*coarse*	0	9.583	112.563	11,75
3	M	*moderate*	0	32.429	412.251	12,71
4	F	*fine*	0	101.296	1.368.594	13,51
5	VC2	*very coarse*	2	271.272	3.686.268	13,59
6	M1	*moderate*	1	297.927	4.134.255	13,88
7	VF	*very fine*	0	658.609	9.294.721	14,11
8	F1	*fine*	1	882.824	12.562.880	14,23
9	C2	*coarse*	2	906.882	12.854.824	14,17
10	VC3	*very coarse*	3	2.382.864	34.128.924	14,32
11	M2	*moderate*	2	2.539.954	36.768.808	14,48
12	VF1	*very fine*	1	5.413.520	78.935.174	14,58

Tabla 4.1: Matrices SDP de prueba escogidas para nuestro estudio experimental.

dificulta el cálculo *manual* de separadores de vértices, motivando el empleo de heurísticos para este propósito en la etapa de particionado del grafo de adyacencia de la matriz del sistema.

- Los sistemas de ecuaciones resultantes presentan propiedades algebraicas para las cuales la jerarquía multinivel construida por el método numérico de ILUPACK se muestra especialmente efectiva para mejorar la eficiencia y escalabilidad del método PCG [31].
- Los coeficientes $K(x, y, z)$ aleatorios en Ω aumentan la irregularidad de los cálculos involucrados en el cálculo paralelo y aplicación paralela del precondicionador, debido a la diferente distribución de las incógnitas aceptadas y rechazadas por el método numérico en cada uno de los subgrafos independientes.
- Al considerar matrices de distintos tamaños (ver Tabla 4.1) y distinto número de procesadores, es posible cuantificar a qué ritmo se reduce la eficiencia paralela de nuestro enfoque cuando aumenta el número de procesadores (para una matriz de tamaño fijo), y en qué medida varía cuando aumenta el tamaño de la matriz (para un número fijo de procesadores).

4.2. Configuración de los algoritmos y entorno de ejecución paralelos

4.2.1. Configuración de los algoritmos paralelos

En este apartado describimos, para el problema de aplicación (4.1), la configuración de cada una de las etapas que constituyen nuestro enfoque para el aprovechamiento de paralelismo multihebra en el precondicionado y en la resolución iterativa de sistemas de ecuaciones lineales dispersos y de gran dimensión. Repasando el desarrollo de los capítulos previos, este enfoque consta de las siguientes tres etapas: la disección anidada para la obtención del árbol de tareas, ilustrada en el Algoritmo 2.20; el cálculo del precondicionador LDL^T multinivel, mostrado en el Algoritmo 2.21; y la resolución iterativa del sistema mediante el método PCG, expresado en el Algoritmo 3.19.

Disección anidada

La implementación del Algoritmo 2.20 corre a cargo de la biblioteca secuencial de particionado de grafos SCOTCH [128], versión 5.0.6RC16. En esta biblioteca, la rutina calcular_separador (línea 2 del Algoritmo 2.20) implementa heurísticos multinivel [89, 102] para el cálculo de separadores de vértices, y no explota paralelismo de ningún tipo; es decir, se trata de una rutina

estrictamente *secuencial*. Aunque los parámetros de los heurísticos multinivel se pueden ajustar para cada problema concreto con el objeto de mejorar el equilibrio entre tiempo de computación, memoria consumida y calidad del separador, está fuera de los objetivos de esta tesis realizar un estudio exhaustivo sobre cuál es el mejor ajuste para cada caso. En su lugar, hemos empleado el valor por defecto que la biblioteca SCOTCH les asigna; para más detalles sobre los parámetros de este tipo de heurísticos y el valor por defecto que la biblioteca SCOTCH les asigna, consultar [130]. En lo que respecta a la numeración de los nodos de los subgrafos independientes, consideramos dos estrategias de reordenación distintas:

- `ND-HAMD-A`: los nodos de los subgrafos independientes se numeran mediante MLND[1] en la etapa inicial de particionado (línea 10 del Algoritmo 2.20). Aunque esta estrategia crea (internamente) una subjerarquía de separadores y subgrafos independientes, el propósito no es aumentar el grado de paralelismo para etapas posteriores, sino calcular una numeración de los nodos de los subgrafos independientes que permita reducir el llenado durante el proceso de factorización. Cuando el número de vértices de los subgrafos independientes de esta subjerarquía es inferior a 120, entonces el proceso recursivo de disección se detiene, y sus vértices se numeran utilizando el algoritmo del mínimo grado aproximado (*Approximate Minimum Degree*, AMD) [14]. La reordenación de los subgrafos independientes se realiza *secuencialmente*. Cabe mencionar los paquetes SCOTCH y METIS [102] emplean por defecto esta estrategia para calcular reordenaciones para reducción de llenado.

- `ND-HAMD-B`: los nodos de los subgrafos independientes se numeran en paralelo durante el cálculo del precondicionador LDL^T multinivel, explotando el paralelismo de tarea. En concreto, cada tarea hoja del árbol aplica una estrategia de reordenación para reducción de llenado en el *primer nivel* del proceso que la tarea realiza localmente (línea 11 del Algoritmo 2.21). Por ejemplo, en el caso de las Figuras 2.14 y 2.15, la tarea $(3,2)$ numera los nodos de $G_{(3,2)}$ aplicando para ello una estrategia de reordenación para reducción de llenado sobre el bloque $A_{22}^{(3,2)}$; las tareas $(3,1)$, $(3,3)$ y $(3,4)$ hacen lo propio con $G_{(3,1)}$, $G_{(3,3)}$ y $G_{(3,4)}$, respectivamente. Aunque esta reordenación se puede llevar a cabo utilizando MLND (como en el caso de `ND-HAMD-A`), consideramos, en su lugar, HAMD (*Halo-AMD*) [131]. HAMD es una variante restringida de AMD que, en lugar de considerar el bloque $A_{22}^{(3,2)}$ de forma aislada (como AMD o MLND), tiene en cuenta las conexiones entre los nodos de los subgrafos independientes y los separadores (es decir, las aristas correspondientes a las entradas de los bloques $A_{55}^{(3,2)}$ y $A_{72}^{(3,2)}$) para calcular los grados de los vértices durante el proceso de factorización simbólica de $A_{22}^{(3,2)}$. Esta propiedad de HAMD permite generalmente mejorar la calidad de la permutación para reducción de llenado [131]. Además, al igual que AMD, HAMD es una estrategia de reordenación para reducción de llenado computacionalmente más ecónomica que MLND [89]. Para más detalles sobre el funcionamiento, coste y calidad de HAMD, consultar [131].

La Figura 4.2 ilustra de forma gráfica estas dos estrategias. Como se puede observar en ella, ambas calculan los separadores *secuencialmente* en la etapa inicial de particionado. Sin embargo, mientras que `ND-HAMD-A` utiliza MLND y no explota el paralelismo de tareas presente en el árbol para numerar los nodos de los subgrafos independientes, `ND-HAMD-B` los numera con HAMD, explotando paralelismo de tareas durante el cálculo del precondicionador LDL^T multinivel.

En la Figura 4.3 (a) mostramos el patrón de dispersidad de una de las matrices de prueba consideradas para nuestro estudio experimental (matriz C, Tabla 4.1). En las Figuras 4.3 (b) y (c) mostramos, a modo de ejemplo, el patrón de dispersidad de la reordenación, $A \rightarrow \Omega^T A \Omega$, derivada de la numeración por niveles construida a partir de las estrategias `ND-HAMD-A` y `ND-HAMD-B` con $l = 3$ niveles de recursividad; ver Figuras 2.11 y 4.2. Las lineas de particionado vertical

[1] En este punto del desarrollo es necesario recordar que MLND (*MultiLevel Nested Dissection*) es la variante de la disección anidada que emplea heurísticos multinivel para el cálculo de separadores de vértices [89, 102].

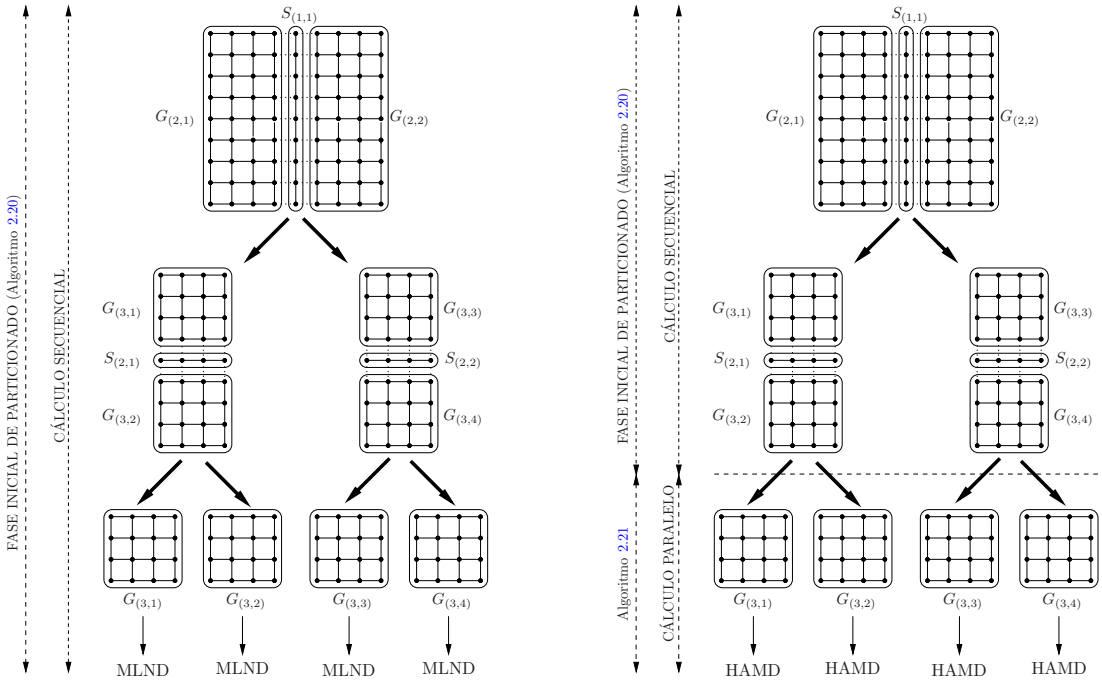


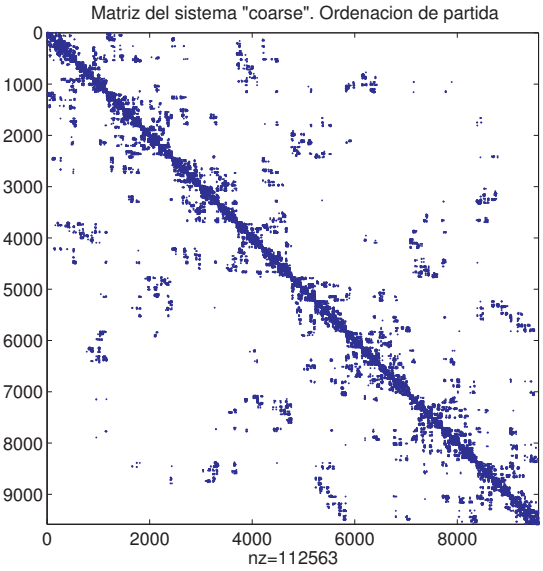
Figura 4.2: `ND-HAMD-A` (izquierda) y `ND-HAMD-B` (derecha).

y horizontal en las Figuras 4.3 (b) y (c) se corresponden con el particionado por bloques de la matriz $\Omega^T A \Omega$ en (2.87). Los tamaños (números de nodos) de los separadores $S_{(1,1)}$, $S_{(2,1)}$ y $S_{(2,2)}$ que calculó el Algoritmo 2.20 fueron 280, 132 y 144, respectivamente, y el de los subgrafos independientes $G_{(3,1)}$, $G_{(3,2)}$, $G_{(3,3)}$ y $G_{(3,4)}$, 2341, 2179, 2497 y 2010, respectivamente.

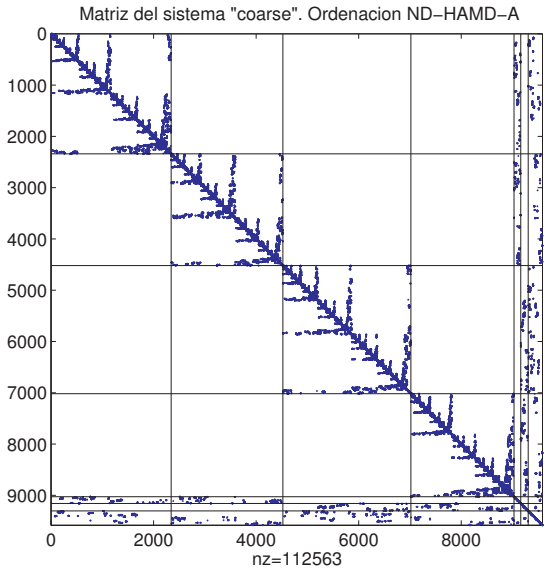
Cálculo del precondicionador LDL^T multinivel

A continuación, describimos la configuración del Algoritmo 2.21, y más concretamente de la rutina calcular_factorización (línea 11), que se encarga de realizar las computacionales locales de la tarea que recibe como parámetro. La configuración de esta rutina involucra la elección de:

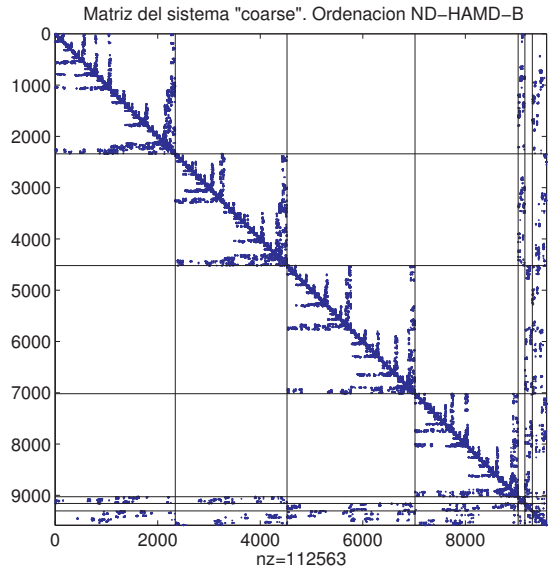
- *Estrategias de reordenación locales.* De acuerdo con el Apartado 2.3.3, nuestro enfoque para el aprovechamiento de paralelismo de tareas acomoda la aplicación de escalados y permutaciones (para reducción de llenado) locales, aunque estas operaciones se deben restringir a las bloques que factoriza la tarea correspondiente. Para explotar esta funcionalidad y tener en cuenta al mismo tiempo este requisito, consideramos la aplicación de HAMD como estrategia de reordenación local en la rutina calcular_factorización. En concreto, la configuración de esta rutina depende del tipo de tarea (tarea hoja o tarea con hijos) y de la estrategia utilizada para la etapa inicial de particionado (`ND-HAMD-A` o `ND-HAMD-B`):
  - Las tareas hoja *no* reordenan la matriz de entrada al primer nivel si la estrategia es `ND-HAMD-A` (ya que ya ha sido reordenada por MLND en la etapa inicial de particionado; ver Figura 4.2 (a)), y la reordenan aplicando HAMD si es `ND-HAMD-B` (ver Figura 4.2 (b)).
  - Las tareas hoja aplican localmente HAMD a partir del segundo nivel (inclusive).
  - Las tareas con hijos aplican localmente HAMD a partir del primer nivel (inclusive).
- *Valor de los parámetros τ y κ.* Estos dos parámetros controlan, respectivamente, las tolerancias de descarte y el crecimiento de la norma de $\tilde{L}^{-1}$ en el proceso de factorización LDL^T con pivotamiento basado en la inversa (consultar la Sección 2.2). Además, determinan el



(a)



(b)



(c)

Figura 4.3: (a) Patrón de dispersidad de la ordenación de partida de la matriz C (ver Tabla 4.1). (b) Patrón de dispersidad de la reordenación derivada de la numeración por niveles construida a partir de la estrategia `ND-HAMD-A` con $l = 3$ niveles de recursividad. (c) Patrón de dispersidad de la reordenación derivada de la numeración por niveles construida a partir de la estrategia `ND-HAMD-B` con $l = 3$ niveles de recursividad.

llenado y, por tanto, el coste computacional y de memoria del cálculo del precondicionador LDL^T multinivel; valores más altos de κ o τ resultan en un precondicionador más económico, aunque (puede que) de peor calidad numérica, mientras que valores más bajos pueden resultar en un precondicionador de alta calidad, aunque muy costoso de calcular, aplicar y almacenar. Evaluando todas las combinaciones posibles de los valores $\kappa = 3, 5, 10$ y $\tau = 5 \times 10^{-1}, 3 \times 10^{-1}, 10^{-1}, 5 \times 10^{-2}, 10^{-2}, 5 \times 10^{-3}, 10^{-3}$, hemos observado que la combinación $\kappa = 5$ y $\tau = 10^{-2}$ logra el mejor compromiso entre memoria y tiempo para muchos de los sistemas considerados en la Tabla 4.1, e incluso un compromiso razonable (en relación al mejor) cuando no lo logra. Por lo tanto, para simplificar la presentación, no afinamos el valor de estos parámetros para obtener el mejor compromiso en cada caso, escogiendo en su lugar $\kappa = 5$ y $\tau = 10^{-2}$ para todos los experimentos recogidos en esta sección.

Resolución iterativa del sistema mediante el método PCG

La configuración del Algoritmo 3.19 involucra la elección de:

- *Algoritmos para la aplicación del precondicionador LDL^T multinivel (líneas 6 y 14).* Según lo especificado en el Apartado 3.4.5, esta operación se puede llevar a cabo mediante los Algoritmos 3.12 y 3.13, o los Algoritmos 3.12 y 3.14. La diferencia entre estas dos alternativas radica en la estrategia que se emplea para determinar el mapeado de tareas a procesos para el proceso de resolución progresiva paralela; mientras que el Algoritmo 3.13 utiliza invariablemente el mapeado que resulta del cálculo del precondicionador, el Algoritmo 3.14 puede realizar dinámicamente ajustes en este mapeado con el objetivo de reducir los tiempos de espera de los procesos. Tras realizar el estudio comparativo de los Algoritmos 3.13 y 3.14, hemos observado que las conclusiones extraídas en el Apartado 3.4.5 se pueden extender a la EDP tridimensional (4.1). En la mayoría de los casos, el rendimiento relativo de ambos algoritmos es comparable y, cuando no lo es, el Algoritmo 3.14 es *generalmente* más rápido que el Algoritmo 3.13. Por tanto, para simplificar la presentación, omitimos el estudio comparativo de ambos algoritmos, y escogemos que la aplicación del precondicionador LDL^T multinivel se lleve a cabo mediante los Algoritmos 3.12 y 3.14.

- *Solución inicial y criterio de parada.* La solución inicial es el vector nulo, y el proceso iterativo se detiene cuando se cumple el criterio de convergencia descrito en [152]. Este criterio emplea una estimación del error relativo en la A-norma, $\|x - x^{(i)}\|_A / \|x - x^{(0)}\|_A$, para evaluar, en cada iteración, la precisión de la solución aproximada calculada hasta el momento. Si $\|x - x^{(i)}\|_A / \|x - x^{(0)}\|_A \lessapprox \epsilon$, donde ϵ es un parámetro predefinido, entonces se detiene el proceso iterativo. Para todos los experimentos de este capítulo hemos escogido $\epsilon = 10^{-8}$, el valor por defecto en la biblioteca ILUPACK.

4.2.2. Configuración del entorno de ejecución paralelo

Los resultados experimentales presentados en este capítulo se han obtenido al ejecutar nuestra implementación de los algoritmos paralelos en el multiprocesador con memoria compartida SGI Altix 350 (consultar el Apéndice B). Los códigos están escritos en los lenguajes C y Fortran77, y utilizan el estándar OpenMP para habilitar el modelo de programación paralela de variables compartidas. La compilación se realizó con la versión 9.0 de los compiladores `icc` e `ifort` de Intel, utilizando `-O3` como *flag* de optimización, y el soporte que estos compiladores dan a la revisión 2.5 del estándar OpenMP. Asimismo, para explotar paralelismo durante la factorización directa LDL^T de matrices densas (requerida potencialmente durante el cálculo del precondicionador LDL^T multinivel), hemos enlazado los códigos con la biblioteca LAPACK y la versión *multihebra* de la biblioteca BLAS incluidas en la versión 10.0 de la biblioteca MKL de Intel. Todas las computaciones numéricas se han efectuado aritmética en coma flotante IEEE 754 de doble

precisión. Por último, hemos configurado el entorno de ejecución paralelo de tal forma que ligamos una hebra por procesador físico del multiprocesador, y evitamos que las hebras migren durante la ejecución de los códigos paralelos. Este último ajuste es conveniente ya que, en el multiprocesador SGI Altix 350, la memoria reservada por una hebra se ubica en el módulo físico de memoria correspondiente al procesador físico en el que se ejecuta dicha hebra. Si el sistema operativo mueve una hebra a un procesador físico situado un nodo remoto durante la ejecución paralela, un alto porcentaje de los accesos a memoria que realiza dicha hebra a partir de ese momento se referirán a posiciones situadas en nodos remotos, pudiendo afectar de forma significativa al rendimiento de los códigos paralelos.

4.3. Evaluación del rendimiento de los algoritmos paralelos

Descripción del experimento y presentación de los resultados

El primer experimento que presentamos en esta sección compara el desempeño del resolutor secuencial incluido en la biblioteca ILUPACK con el de nuestro resolutor paralelo, utilizando distinto número de procesadores y dos estrategias alternativas para la etapa inicial de particionado, `ND-HAMD-A` y `ND-HAMD-B` (consultar el Apartado 4.2.1). Para ampliar el rango de la comparación, también consideramos dos alternativas para configurar los heurísticos de reducción de llenado empleados por el precondicionador *secuencial* de la biblioteca ILUPACK (paso 2 del Algoritmo 2.15). La primera, a la que también denominamos como `ND-HAMD-A`, aplica MLND en el primer nivel y, a partir del segundo, AMD; la segunda, a la que nos referimos como `ND-HAMD-B`, aplica AMD a partir del primer nivel. (Estas dos alternativas se derivan naturalmente considerando el algoritmo secuencial como una variante particular del paralelo con un árbol de tareas de un sólo nivel/hoja.) La altura (número de niveles) y el número de hojas del árbol de tareas empleado por el resolutor paralelo se fijaron durante la etapa inicial de particionado como una función del parámetro f (ver Algoritmo 2.20), y las mediciones se realizaron conforme a la siguiente estrategia. Para cada combinación de matriz, número de procesadores y valor de f, ejecutamos diez veces la terna formada por las tres etapas del resolutor paralelo. En cada una de estas ejecuciones, registramos el tiempo de ejecución total (es decir, la suma de los tiempos de ejecución de las tres etapas), constituyendo una muestra con 10 resultados. Finalmente, se determina la ejecución correspondiente a la mediana de la muestra, considerando únicamente las mediciones tomadas en dicha ejecución. El experimento explora diferentes valores de f en el intervalo $[p, 3p]$. Sin embargo, con valores de $f > \frac{3}{2}p$, y las matrices de prueba consideradas, la estrategia de planificación dinámica que incorpora el Algoritmo 2.21 no redujo de forma significativa el tiempo de ejecución del resolutor paralelo. Para simplificar la presentación, únicamente mostraremos los resultados obtenidos con un *único* valor de f en el intervalo $[p, \frac{3}{2}p]$, en particular aquel que redujo en mayor medida la suma de los tiempos de ejecución del cálculo del precondicionador multinivel y de la resolución iterativa del sistema, es decir, afinamos el valor del parámetro f para cada matriz y número de procesadores con el objetivo de minimizar el tiempo de ejecución de las etapas numéricas del resolutor.

En las Tablas 4.2, 4.3 y 4.4 comparamos el rendimiento del resolutor secuencial incluido en la biblioteca ILUPACK (resultados para $p = 1$ en las tablas) con el de nuestro resolutor paralelo utilizando $p = 2$, 4, 8 y 16 procesadores (hebras) para la resolución de los 12 sistemas de la Tabla 4.1. En concreto, la Tabla 4.2 presenta los resultados para los sistemas de menor tamaño (VC, C, M y F), la Tabla 4.3, para los de tamaño intermedio (VC2, M1, VF y F1) y, por último, la Tabla 4.4, para los de mayor tamaño (C2, VC3, M2 y VF1). Los resultados para cada matriz se separan en dos grupos (de cinco filas cada uno), en función de la estrategia utilizada para la etapa inicial de particionado. La columna etiquetada como "Total" muestra el tiempo de ejecución total del resolutor, y las tablas muestran resultados separados para cada una de las tres etapas.

Cód.	p	Total	Disecc. Anid.		Precond. LDL^T Multiniv.			Método PCG			
		T_p (seg.)	Opc. ND-HAMD-X	T (seg.)	nnz_M (mill.)	T_p (seg.)	S_p (T_1/T_p)	#Iter.	Err.	T_p (seg.)	S_p (T_1/T_p)
VC	1	0,04	A	0,02	0,0	0,01	1,0	12	4,6e-16	0,01	1,0
	2	0,03	A	0,02	0,0	0,01	1,9	12	1,9e-15	0,01	1,3
	4	0,03	A	0,02	0,0	0,01	1,9	12	1,9e-14	0,01	0,9
	8	0,03	A	0,02	0,0	0,01	2,2	12	2,8e-14	0,01	0,7
	16	0,04	A	0,02	0,0	0,01	1,5	12	7,0e-14	0,02	0,5
	1	0,02	B	0,00	0,0	0,02	1,0	11	2,3e-15	0,01	1,0
	2	0,02	B	0,00	0,0	0,01	1,7	11	2,2e-14	0,01	1,3
	4	0,03	B	0,01	0,0	0,01	2,1	12	3,2e-15	0,01	0,7
	8	0,03	B	0,01	0,0	0,01	2,4	12	5,9e-15	0,01	0,6
	16	0,04	B	0,01	0,0	0,01	1,7	12	4,2e-14	0,02	0,4
C	1	0,31	A	0,15	0,1	0,11	1,0	15	4,6e-13	0,06	1,0
	2	0,24	A	0,15	0,1	0,06	1,9	15	6,3e-13	0,03	1,7
	4	0,21	A	0,15	0,1	0,04	3,0	15	1,3e-12	0,02	2,4
	8	0,20	A	0,15	0,1	0,03	3,7	15	2,2e-12	0,03	2,1
	16	0,20	A	0,15	0,1	0,03	4,3	15	3,7e-12	0,03	1,9
	1	0,18	B	0,00	0,1	0,13	1,0	14	8,2e-13	0,05	1,0
	2	0,12	B	0,02	0,1	0,07	1,9	14	1,4e-12	0,03	1,8
	4	0,11	B	0,04	0,1	0,04	3,0	15	5,8e-13	0,02	2,2
	8	0,13	B	0,07	0,1	0,03	3,6	15	8,8e-13	0,03	1,9
	16	0,15	B	0,09	0,1	0,03	4,3	15	1,8e-12	0,03	1,7
M	1	1,45	A	0,60	0,4	0,52	1,0	18	5,1e-12	0,33	1,0
	2	1,02	A	0,60	0,5	0,27	2,0	18	5,9e-12	0,16	2,1
	4	0,84	A	0,60	0,5	0,15	3,5	18	1,3e-11	0,09	3,6
	8	0,77	A	0,60	0,5	0,10	5,3	18	2,3e-11	0,07	4,5
	16	0,75	A	0,60	0,5	0,09	6,1	18	2,4e-11	0,07	5,0
	1	0,89	B	0,00	0,4	0,60	1,0	17	8,5e-12	0,30	1,0
	2	0,59	B	0,13	0,4	0,31	1,9	18	6,2e-12	0,15	2,0
	4	0,44	B	0,18	0,4	0,17	3,6	18	1,2e-11	0,09	3,3
	8	0,43	B	0,25	0,4	0,11	5,4	18	2,2e-11	0,07	4,2
	16	0,42	B	0,26	0,4	0,10	6,1	18	2,8e-11	0,06	4,6
F	1	6,34	A	2,32	1,6	2,26	1,0	22	9,5e-11	1,76	1,0
	2	4,36	A	2,32	1,6	1,15	2,0	23	1,1e-10	0,89	2,0
	4	3,38	A	2,33	1,6	0,61	3,7	23	6,9e-11	0,45	4,0
	8	2,98	A	2,32	1,6	0,38	5,9	24	9,1e-11	0,28	6,4
	16	2,78	A	2,32	1,6	0,26	8,7	24	1,1e-10	0,20	8,7
	1	4,39	B	0,00	1,5	2,61	1,0	23	5,0e-11	1,77	1,0
	2	2,54	B	0,39	1,5	1,29	2,0	23	8,2e-11	0,87	2,1
	4	1,70	B	0,56	1,5	0,70	3,8	23	9,9e-11	0,44	4,1
	8	1,39	B	0,70	1,5	0,42	6,2	23	1,9e-10	0,26	6,8
	16	1,39	B	0,91	1,5	0,29	9,1	23	2,6e-10	0,19	9,3

Tabla 4.2: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Los resultados de esta tabla se refieren a los cuatro sistemas de menor tamaño, es decir, VC, C, M y F en la Tabla 4.1.

Desempeño de la disección anidada multinivel

Las dos columnas etiquetadas como "Disecc. Anid." en las Tablas 4.2, 4.3 y 4.4, es decir, "Opc." y "T", muestran, respectivamente, la estrategia usada para la disección anidada (es decir, `ND-HAMD-A` o `ND-HAMD-B`), y su tiempo de ejecución en segundos. La ejecución de esta etapa es completamente secuencial (la biblioteca SCOTCH no explota paralelismo de ningún tipo durante el proceso). Por lo tanto, para esta etapa, p no es realmente el número de procesadores empleados en su ejecución, sino que se utiliza únicamente para ajustar el grado de paralelismo disponible para el cálculo del precondicionador LDL^T multinivel y la resolución iterativa del sistema. Como se puede observar en las tablas, el tiempo de ejecución de la etapa inicial de particionado

Cód.	p	Total	Disecc. Anid.		Precond. LDL^T Multiniv.			Método PCG			
		T_p (seg.)	Opc. ND-HAMD-X	T (seg.)	nnz_M (mill.)	T_p (seg.)	S_p (T_1/T_p)	#Iter.	Err.	T_p (seg.)	S_p (T_1/T_p)
VC2	1	21,56	A	6,94	4,6	7,19	1,0	29	3,7e-10	7,43	1,0
	2	13,91	A	6,93	4,5	3,52	2,0	30	2,9e-10	3,46	2,1
	4	10,41	A	6,93	4,6	1,85	3,9	29	3,9e-10	1,63	4,6
	8	9,04	A	6,94	4,6	1,07	6,7	30	3,6e-10	1,03	7,2
	16	8,25	A	6,94	4,6	0,67	10,7	30	4,4e-10	0,63	11,7
	1	16,07	B	0,00	4,5	9,00	1,0	28	4,3e-10	7,07	1,0
	2	8,26	B	1,08	4,5	4,03	2,2	28	6,9e-10	3,16	2,2
	4	5,23	B	1,58	4,5	2,05	4,4	29	4,2e-10	1,61	4,4
	8	4,16	B	1,96	4,5	1,21	7,4	29	4,2e-10	0,99	7,1
	16	3,89	B	2,52	4,5	0,74	12,1	30	4,6e-10	0,63	11,2
M1	1	24,18	A	8,01	5,0	7,93	1,0	29	1,7e-10	8,25	1,0
	2	15,63	A	8,01	5,1	3,94	2,0	29	2,7e-10	3,67	2,2
	4	12,00	A	8,01	5,1	2,12	3,7	29	2,7e-10	1,87	4,4
	8	10,44	A	8,01	5,1	1,27	6,2	29	5,6e-10	1,16	7,1
	16	9,56	A	8,02	5,1	0,79	10,0	30	2,8e-10	0,76	10,9
	1	17,44	B	0,00	4,7	9,77	1,0	28	1,3e-10	7,66	1,0
	2	8,98	B	1,19	4,8	4,41	2,2	28	3,0e-10	3,38	2,3
	4	5,82	B	1,71	4,8	2,36	4,1	28	5,0e-10	1,75	4,4
	8	4,68	B	2,12	4,8	1,42	6,9	29	3,1e-10	1,14	6,7
	16	4,05	B	2,43	4,8	0,91	10,8	28	4,0e-10	0,71	10,8
VF	1	62,56	A	19,51	11,3	19,30	1,0	35	7,4e-10	23,76	1,0
	2	40,63	A	19,51	11,4	9,69	2,0	36	5,2e-10	11,43	2,1
	4	30,08	A	19,54	11,4	5,06	3,8	36	9,1e-10	5,48	4,3
	8	25,74	A	19,53	11,4	3,00	6,4	36	1,1e-09	3,21	7,4
	16	23,05	A	19,54	11,5	1,64	11,8	36	1,1e-09	1,87	12,7
	1	47,25	B	0,01	10,8	24,82	1,0	34	5,8e-10	22,43	1,0
	2	24,74	B	2,79	11,0	11,53	2,2	34	8,2e-10	10,42	2,2
	4	14,73	B	4,00	11,0	5,74	4,3	34	9,2e-10	4,99	4,5
	8	11,22	B	4,81	11,0	3,37	7,4	35	9,5e-10	3,03	7,4
	16	10,02	B	6,39	11,1	1,82	13,6	35	9,2e-10	1,81	12,4
F1	1	90,35	A	27,04	15,7	27,12	1,0	39	1,5e-09	36,20	1,0
	2	59,17	A	27,04	15,9	14,14	1,9	39	1,4e-09	17,99	2,0
	4	43,33	A	27,04	15,9	7,33	3,7	40	8,3e-10	8,96	4,0
	8	35,20	A	27,05	15,9	3,77	7,2	40	1,3e-09	4,38	8,3
	16	32,05	A	27,01	15,9	2,24	12,1	40	1,2e-09	2,81	12,9
	1	69,65	B	0,01	15,2	36,03	1,0	37	9,9e-10	33,62	1,0
	2	37,90	B	3,61	14,9	17,28	2,1	38	1,6e-09	17,01	2,0
	4	21,33	B	4,86	14,9	8,30	4,3	38	1,2e-09	8,17	4,1
	8	15,09	B	6,77	15,0	4,24	8,5	38	1,7e-09	4,08	8,3
	16	13,59	B	8,46	15,0	2,48	14,6	39	1,1e-09	2,65	12,7

Tabla 4.3: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Los resultados de esta tabla se refieren a los cuatro sistemas de tamaño intermedio, es decir, VC2, M1, VF y F1 en la Tabla 4.1.

es independiente de p en el caso de ND-HAMD-A. Esta estrategia numera los nodos de los subgrafos independientes mediante MLND, con el objetivo de reducir el llenado que se produce durante el proceso de factorización y, por tanto, su coste no depende de p, sino del umbral superior que acota el tamaño de los subgrafos independientes (120 nodos en todos los experimentos) de la subjerarquía construida internamente para calcular esta numeración (consultar el Apartado 4.2.1 para más detalles). Comparando el tiempo de ejecución de esta etapa con el tiempo de ejecución total, podemos observar que su peso relativo crece con p y, además, acaba dominando casi por completo el coste de todo el proceso. Por ejemplo, en el caso de la matriz C2, este peso fue, para $p = 2$, 4, 8, y 16 procesadores, del 45 %, 63 %, 70 % y 83 %, respectivamente; se pueden observar

Cód.	p	Total T_p (seg.)	Disecc. Anid. Opc. ND-HAMD-X	Disecc. Anid. T (seg.)	Precond. LDL^T Multiniv. nnz_M (mill.)	Precond. LDL^T Multiniv. T_p (seg.)	Precond. LDL^T Multiniv. S_p (T_1/T_p)	Método PCG #Iter.	Método PCG Err.	Método PCG T_p (seg.)	Método PCG S_p (T_1/T_p)
C2	1	94,81	A	27,80	16,1	27,96	1,0	41	2,0e-09	39,06	1,0
	2	60,71	A	27,81	16,2	14,11	2,0	41	2,2e-09	18,79	2,1
	4	43,91	A	27,80	16,3	7,12	3,9	41	1,9e-09	8,98	4,3
	8	36,25	A	27,80	16,4	3,87	7,2	41	2,3e-09	4,57	8,5
	16	33,16	A	27,79	16,4	2,42	11,6	42	1,8e-09	2,95	13,2
	1	73,63	B	0,01	15,9	35,77	1,0	40	1,9e-09	37,85	1,0
	2	39,66	B	4,08	16,1	17,38	2,1	40	2,1e-09	18,20	2,1
	4	22,98	B	5,73	16,1	8,53	4,2	40	1,8e-09	8,72	4,3
	8	16,45	B	7,39	16,2	4,59	7,8	40	1,9e-09	4,47	8,5
	16	14,25	B	8,69	16,2	2,70	13,2	40	2,4e-09	2,87	13,2
VC3	1	310,23	A	82,14	45,0	90,83	1,0	52	4,0e-09	137,25	1,0
	2	190,33	A	82,13	45,2	40,77	2,2	52	4,2e-09	67,43	2,0
	4	136,34	A	82,11	45,2	20,55	4,4	52	4,0e-09	33,68	4,1
	8	110,63	A	82,13	44,9	11,18	8,1	52	5,2e-09	17,32	7,9
	16	98,76	A	82,15	44,9	6,69	13,6	53	3,7e-09	9,92	13,8
	1	247,07	B	0,02	45,2	114,30	1,0	50	4,8e-09	132,75	1,0
	2	129,14	B	12,58	45,2	51,80	2,2	50	4,1e-09	64,77	2,0
	4	72,85	B	14,83	45,2	25,48	4,5	50	4,0e-09	32,53	4,1
	8	52,29	B	21,52	45,0	13,47	8,5	52	4,1e-09	17,30	7,7
	16	42,52	B	25,18	44,9	7,48	15,3	52	4,5e-09	9,86	13,5
M2	1	327,17	A	86,96	47,8	98,29	1,0	50	3,7e-09	141,93	1,0
	2	203,87	A	86,96	47,7	44,51	2,2	51	2,9e-09	72,40	2,0
	4	144,03	A	86,97	47,6	21,94	4,5	51	3,4e-09	35,12	4,0
	8	116,91	A	86,95	47,6	11,85	8,3	51	3,1e-09	18,11	7,8
	16	103,38	A	87,02	47,7	6,51	15,1	52	2,6e-09	9,85	14,4
	1	262,06	B	0,03	47,7	122,33	1,0	50	2,1e-09	139,70	1,0
	2	135,09	B	9,08	47,3	55,55	2,2	50	2,8e-09	70,45	2,0
	4	77,81	B	16,21	47,1	26,78	4,6	51	2,5e-09	34,83	4,0
	8	52,86	B	20,83	47,2	14,03	8,7	51	2,6e-09	18,01	7,8
	16	42,41	B	25,45	47,3	7,36	16,6	51	2,7e-09	9,59	14,6
VF1	1	977,34	A	221,86	105,4	244,73	1,0	64	4,8e-09	510,75	1,0
	2	588,87	A	221,50	105,2	117,67	2,1	62	6,6e-09	249,70	2,0
	4	374,43	A	221,49	104,7	51,42	4,8	64	5,8e-09	101,52	5,0
	8	299,75	A	221,42	104,3	26,79	9,1	63	8,1e-09	51,54	9,9
	16	265,43	A	221,45	104,0	14,81	16,5	64	6,4e-09	29,17	17,5
	1	705,13	B	0,05	100,7	287,40	1,0	60	7,7e-09	417,68	1,0
	2	358,72	B	30,98	98,8	134,77	2,1	62	5,6e-09	192,97	2,2
	4	201,32	B	44,62	98,2	64,42	4,5	61	7,4e-09	92,28	4,5
	8	140,41	B	58,25	98,4	32,98	8,7	62	6,4e-09	49,18	8,5
	16	115,89	B	70,26	99,0	17,69	16,2	62	6,8e-09	27,94	15,0

Tabla 4.4: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Los resultados de esta tabla se refieren a los cuatro sistemas de mayor tamaño, es decir, C2, VC3, M2 y VF1 en la Tabla 4.1.

pesos relativos similares para las matrices de tamaño intermedio (Tabla 4.3) y para el resto de matrices de mayor tamaño (Tabla 4.4). Por otra parte, el tiempo de ejecución de la etapa inicial de particionado utilizando `ND-HAMD-B` es inferior al de `ND-HAMD-A`. Sin embargo, *aumenta con el número de procesadores y acaba dominando el tiempo de ejecución total (aunque en menor grado que* `ND-HAMD-A`*)*. `ND-HAMD-B` trunca el proceso recursivo de particionado tan pronto como se extrae el suficiente paralelismo para p procesadores. Un valor más alto de p se traduce en niveles adicionales de recursividad en la disección anidada de $G(A)$ y, por tanto, el coste de este proceso aumenta. Como se puede observar en las tablas, el peso relativo de `ND-HAMD-B` con respecto al de `ND-HAMD-A` crece de forma considerable con p para el número de procesadores

considerados. Por ejemplo, en el caso de la matriz M1 (Tabla 4.3), este peso es, para $p = 2$, $p = 4$, $p = 8$, $p = 16$ procesadores, del 15 %, 21 %, 26 % y 30 %, respectivamente. Para problemas tridimensionales, es conocido que el coste de calcular los separadores de los primeros niveles (es decir, el coste de `ND-HAMD-B`) concentra una parte significativa del coste de `ND-HAMD-A`. Esto implica que el volumen de paralelismo de tareas que se puede explotar reordenando los subgrafos independientes en paralelo durante el cálculo del precondicionador LDL^T multinivel disminuye rápidamente con p, motivando el uso de soluciones paralelas para el cálculo de separadores de vértices en la etapa inicial de particionado; en la Sección 4.5 consideramos el uso de este tipo de soluciones.

Desempeño del cálculo paralelo del precondicionador LDL^T multinivel

Las columnas etiquetadas como "Precond. LDL^T Multiniv." en las Tablas 4.2, 4.3 y 4.4 muestran el rendimiento del cálculo paralelo del precondicionador LDL^T multinivel. La columna etiquetada como "nnz_M" se refiere al número de millones de elementos no nulos del precondicionador. Este valor proporciona una estimación del número de operaciones en coma flotante que son necesarias para aplicar el precondicionador en la etapa de resolución del sistema. En el caso del algoritmo secuencial ($p = 1$), el número de elementos no nulos del precondicionador se obtiene de forma acorde a (2.124). La expresión correspondiente al algoritmo paralelo es una generalización de (2.124) que tiene en cuenta la factorización LDL^T multinivel que cada tarea del árbol calcula localmente. Como se puede observar en las tablas, el número de elementos no nulos del precondicionador paralelo sólo varía ligeramente con p y, además, *es cercano al número de elementos no nulos del precondicionador construido por el algoritmo secuencial*. Por ejemplo, en el caso de la matriz M2 y `ND-HAMD-A` (Tabla 4.4), el número de millones de elementos no nulos del precondicionador secuencial fue de 47,8 y, en el caso paralelo, 47,7, 47,6, 47,6, 47,7 para $p = 2$, 4, 8, y 16 procesadores, respectivamente. Esta ligera variación no es sorprendente teniendo en cuenta que, en nuestro enfoque, la extracción de paralelismo de tareas conlleva un cambio del orden de eliminación (consultar el Algoritmo 2.19) y, por tanto, los precondicionadores calculados por el algoritmo con distinto número de procesadores no son exactamente iguales. En las tablas también se puede observar que, con `ND-HAMD-B`, el número de elementos no nulos del precondicionador fue ligeramente inferior que con `ND-HAMD-A` (y cómo veremos más adelante, la calidad del precondicionador también fue inferior). Para los métodos directos de resolución y problemas tridimensionales, la estrategia de `ND-HAMD-A` es cuasi-óptima y típicamente supera a `ND-HAMD-B` [131] (ambas afirmaciones tomando como criterio el coste computacional y de almacenamiento de la factorización directa del sistema reordenado). Sin embargo, en el caso de los métodos iterativos precondicionados mediante factorizaciones LDL^T incompletas, el rendimiento relativo de ambas estrategias depende del problema concreto (consultar el Apartado 2.1.3).

Las tablas también recogen el tiempo de ejecución paralelo en segundos del cálculo del precondicionador (columna etiquetada como "T_p") y la aceleración paralela o *speed-up* (columna etiquetada como "S_p"). Este último indicador cuantifica el beneficio que supone el aprovechamiento de paralelismo, y se obtiene como el cociente entre el tiempo de ejecución paralelo y el tiempo de ejecución del algoritmo secuencial configurado mediante la estrategia de reordenación correspondiente (`ND-HAMD-A` o `ND-HAMD-B`). Para un número determinado de procesadores, el tiempo de ejecución paralelo del cálculo del precondicionador fue siempre inferior en el caso de `ND-HAMD-A` que en el de `ND-HAMD-B`. Esto es así ya que, en el primer caso, las tareas hoja *no* reordenan la matriz de entrada al primer nivel mientras que, en el segundo, la reordenan aplicando HAMD como estrategia para reducción de llenado (consultar el Apartado 4.2.1). Por otra parte, considerando el rendimiento de nuestro enfoque de paralelización, podemos observar que, excepto en el caso del sistema VC (el de menor dimensión), *la aceleración siempre aumenta con* p, independientemente de la estrategia utilizada para la etapa inicial de particionado. Por ejemplo, en el caso del sistema F y `ND-HAMD-A`, el enfoque de paralelización obtiene aceleraciones de 2,0, 3,7, 5,9 y 8,7, para $p = 2$, 4, 8 y 16 procesadores, respectivamente, y de 2,0, 3,8, 6,2, 9,1

con ND-HAMD-B para el mismo número de procesadores. Además, al aumentar el tamaño del sistema (para un número fijo de procesadores) la aceleración (generalmente) aumenta, obteniendo incluso aceleraciones superlineales en algunos casos (consultar, por ejemplo, la Tabla 4.4). Esta última observación evidencia que los sobrecostes asociados al paralelismo se reducen con el tamaño del sistema, aumentando la eficiencia paralela del enfoque de paralelización. Por ejemplo, al aumentar el tamaño del sistema, el coste relativo de las fases en las que se puede explotar un menor grado de concurrencia (ver Algoritmos 2.17 y 2.18) se reduce en relación al coste de la primera fase (ver Algoritmo 2.16), reduciéndose en consecuencia el sobrecoste asociado a los tiempos de espera de los procesadores. La causa de las aceleraciones superlineales puede atribuirse a un factor algorítmico: la diferente distribución de los nodos entre los niveles algebraicos construidos por el método numérico al aumentar el número de procesadores. La parte del sistema asociado a los separadores, es decir, el bloque A_Z en el Algoritmo 2.19, aumenta con p. El algoritmo paralelo *no* procesa esta parte del sistema hasta eliminar la gran mayoría de las filas y columnas correspondientes al bloque A_X, mientras que el algoritmo secuencial lo factoriza desde el primer nivel algebraico, pudiendo rechazar parte de la incógnitas de este bloque en este nivel y en niveles subsecuentes (con el sobrecoste que esto conlleva). Además, también podemos observar que la aceleración alcanzada por el enfoque con ND-HAMD-B fue, en la mayoría de los casos, superior al alcanzado con ND-HAMD-A. De hecho, las aceleraciones superlineales son más frecuentes con ND-HAMD-B que con ND-HAMD-A (comparar, en la Tabla 4.3, las aceleraciones alcanzadas por ambas estrategias.) En el caso de ND-HAMD-B, el algoritmo secuencial reordena la matriz del sistema completo en el primer nivel utilizando AMD, mientras que, en el algoritmo paralelo, las tareas hoja sólo reordenan los bloques correspondientes a los subgrafos independientes en el primer nivel local utilizando HAMD. Un perfil (*profile*) detallado del rendimiento del algoritmo paralelo, reveló que el coste agregado de la reordenación de los subgrafos independientes disminuye al aumentar el número de procesadores, por lo que el rendimiento del algoritmo paralelo con ND-HAMD-B se beneficia de este factor. Aparte de estos factores algorítmicos, también pueden contribuir otros factores arquitecturales, como el mejor aprovechamiento de la localidad de referencia en las cachés de los procesadores provocado por la reducción del tamaño de las estructuras de datos locales de las tareas al aumentar el número de procesadores.

Desempeño de la resolución iterativa paralela del sistema precondicionado

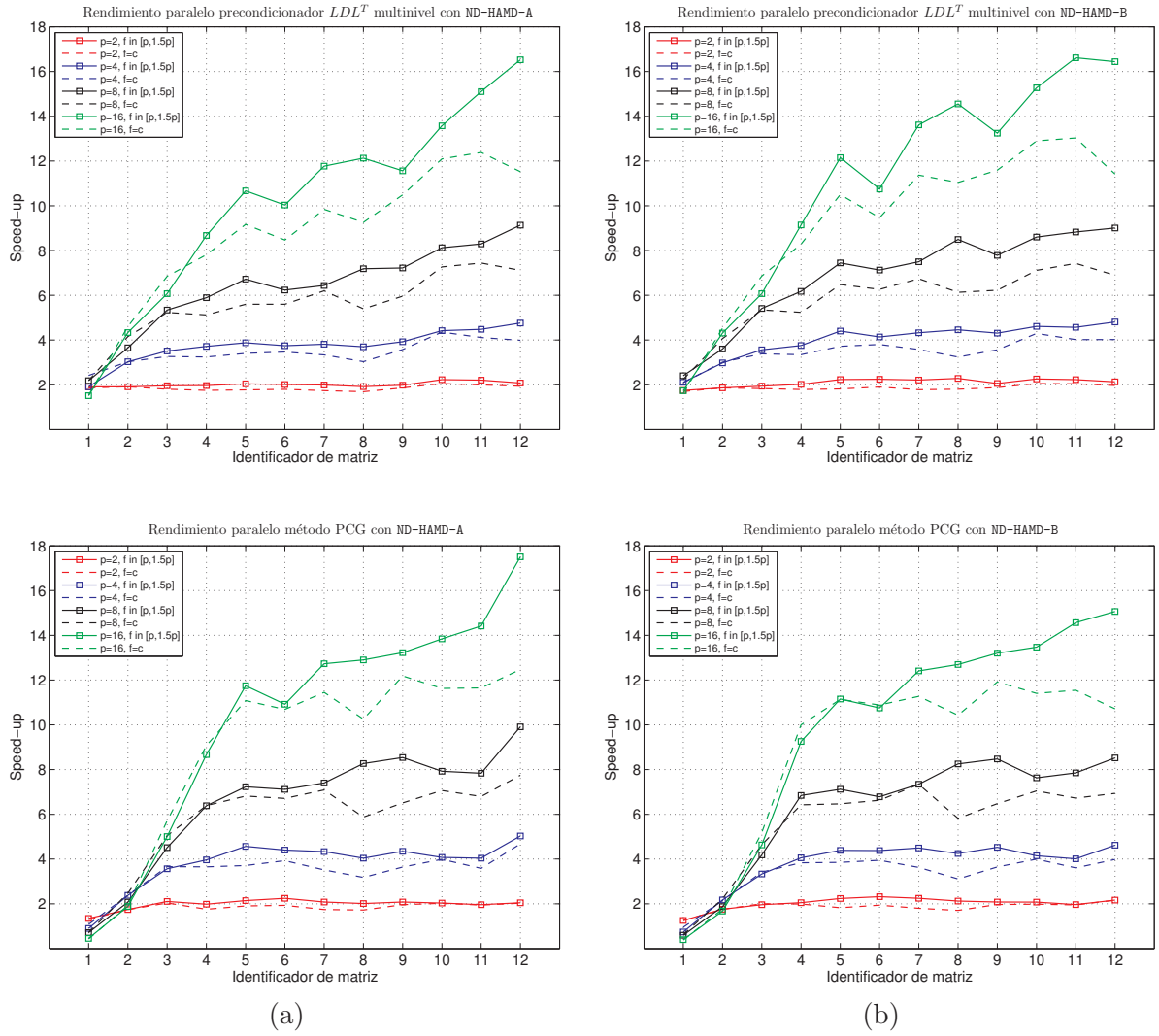
Finalmente, en las columnas etiquetadas como "Método PCG" en las Tablas 4.2, 4.3 y 4.4 ofrecen el rendimiento de la resolución iterativa del sistema. Las columnas etiquetadas como "#Iter." y "Err." muestran, respectivamente, el número de iteraciones requeridas por el Algoritmo 3.19 para converger a la solución de (3.79), y el *error relativo* en la norma-2 vectorial correspondiente a la solución aproximada calculada por el método PCG (ver línea 23 del Algoritmo 3.19). Este último indicador, que se ha podido calcular ya que la solución exacta es conocida, mide, a posteriori, la *precisión* de la solución calculada por el método iterativo de resolución. Como se puede observar en las tablas, el número de iteraciones requeridas por el método y la precisión de la solución calculada por éste, sólo varían ligeramente con p y, además, *esta última es cercana a la precisión de la solución calculada por el método secuencial en todos los casos.* Este resultado evidencia que las transformaciones que realiza nuestro enfoque para la extracción de paralelismo de tareas (consultar el Apartado 2.3.2) no alteran la semántica (entendida como las propiedades numéricas) del algoritmo básico de precondicionado de la biblioteca ILUPACK; en el Apartado 4.4 proporcionamos evidencias adicionales que refuerzan esta afirmación. Las tablas también muestran el tiempo de ejecución paralelo en segundos de la resolución del sistema (columna etiquetada como "T_p") y la aceleración paralela o *speed-up* (columna etiquetada como "S_p") obtenida por nuestro enfoque para esta etapa. Para un número dado de procesadores, el tiempo de ejecución del método PCG es (ligeramente) inferior con ND-HAMD-B que con ND-HAMD-A. Esto debe a que el número de elementos no nulos del precondicionador (y, por tanto,

el número de operaciones necesarias para aplicarlo) y el número de iteraciones requeridas por el método son (ligeramente) inferiores con `ND-HAMD-B`. Por otra parte, considerando el rendimiento paralelo del enfoque, podemos observar que, excepto en el caso de los sistemas VC y C (los dos sistemas de menor dimensión), *la aceleración siempre aumenta con* p, independientemente de la estrategia utilizada para la etapa inicial de particionado. Además, al aumentar el tamaño de la matriz (para un número fijo de procesadores), la aceleración (generalmente) aumenta. Las aceleraciones superlineales que se obtienen para algunos sistemas (ver, por ejemplo, aceleraciones para C2 y $p = 2$, 4 y 8 procesadores en la Tabla 4.4) creemos que se deben (principalmente) al mejor aprovechamiento de la localidad de referencia en la caché de los procesadores, ya que las operaciones involucradas en la resolución del sistema son operaciones limitadas por el acceso a memoria y, por tanto, más sensibles a este efecto. Las aceleraciones superlineales obtenidas en el caso de la matriz VF1, `ND-HAMD-A`, $p = 4$ (5,5), 8 (9,0) y 16 (17,5) procesadores, son más altas que las obtenidos para otras matrices. Para esta combinación de sistema y estrategia de reordenación, la memoria consumida por el algoritmo secuencial de ILUPACK para calcular el precondicionador es superior a la memoria física disponible en un nodo del multiprocesador SGI Altix 350 y, por tanto, los accesos a memoria a nodos remotos penalizan el rendimiento del algoritmo secuencial. Cuando se emplean $p = 4$, 8 o 16 procesadores, la memoria física disponible aumenta y, además, las estructuras de datos de las tareas son más pequeñas, por lo que caben en las memorias locales de los nodos.

Impacto de la planificación dinámica de tareas

El objetivo del siguiente experimento es cuantificar el beneficio que supone el empleo de la estrategia de planificación dinámica del Algoritmo 2.21 sobre las prestaciones paralelas (*speed-up*) del cálculo del precondicionador y la resolución iterativa del sistema. Para ello, evaluamos el rendimiento de estas dos etapas utilizando dos alternativas para ajustar el número de niveles/hojas del árbol de tareas en la etapa de particionado. La primera, a la que nos referiremos como "$f = c$", obtiene árboles de tareas de altura $log_2(p) + 1$ *completos*, es decir, con p hojas. La segunda obtiene árboles de tareas mediante el Algoritmo 2.20 (con valores de f en el intervalo $[p, \frac{3}{2}p]$). Esta segunda alternativa produce árboles de tareas con más tareas hoja que procesadores, aumentando la probabilidad de que la estrategia de planificación dinámica del Algoritmo 2.21 mejore el equilibrio de la carga computacional. La Figura 4.4 compara, para $p = 2$ (curvas en rojo), 4 (curvas en azul), 8 (curvas en negro) y 16 (curvas en verde) procesadores, y los doce sistemas de la Tabla 4.1 (identificados por el valor correspondiente del eje de abscisas), la aceleración obtenida en el cálculo del precondicionador (parte superior) y en la resolución iterativa del sistema (parte inferior) utilizando $f = c$ (curvas discontinuas) o f en el intervalo $[p, \frac{3}{2}p]$ (curvas continuas). La Figura 4.4 (a) presenta la comparación cuando se utiliza `ND-HAMD-A` como estrategia para la etapa inicial de particionado, y la Figura 4.4 (b) cuando se utiliza `ND-HAMD-B`. Como se puede observar en estas figuras, en el caso de los sistemas de menor tamaño (código $= 1, 2, 3, 4$), la estrategia de planificación dinámica *no* mejora el rendimiento para alguna combinación de estrategia, etapa y número de procesadores, e incluso lo puede empeorar ligeramente. Por ejemplo, en el caso de `ND-HAMD-B`, el sistema M (código = 3 en el eje de abscisas), y $p = 16$, la aceleración se redujo desde 6,9 ($f = c$) hasta 6,1 ($f \in [p, \frac{3}{2}p]$) para el cálculo del precondicionador (parte superior de la Figura 4.4 (b)). Para los sistemas de menor tamaño, el mayor grado de homogeneidad en la distribución de la carga logrado por la estrategia de planificación dinámica al aumentar el número de niveles/hojas del árbol de tareas, no siempre se amortiza, por la penalización asociada a otros factores de sobrecoste. Sin embargo, al aumentar el tamaño del sistema, el peso relativo de estos últimos se reduce, y, por tanto, el grado de heterogeneidad/homegenidad de la distribución de la carga se convierte en un factor de mayor peso en las prestaciones. De hecho, para los sistemas de tamaño intermedio y de mayor tamaño, la estrategia de planificación dinámica siempre mejora el rendimiento paralelo de ambas etapas, especialmente con $p = 8$ y 16 procesadores. Por ejemplo, en el caso de `ND-HAMD-A`, el sistema

F1 (código = 8 en el eje de abscisas), y $p = 16$ procesadores, la aceleración aumentó desde 9,3 hasta 12,1 para el cálculo del precondicionador (parte superior de la Figura 4.4 (a)), y desde 10,3 hasta 12,9 para la resolución iterativa del sistema (parte inferior de la Figura 4.4 (a)); el lector puede observar mejoras similares en el caso de `ND-HAMD-B`. Estas mejoras se pueden atribuir (principalmente) al mayor grado de homegenidad en la distribución de la carga logrado por la estrategia de planificación dinámica al aumentar el número de niveles/hojas del árbol de tareas.



(a) (b)

Figura 4.4: Rendimiento paralelo del cálculo del precondicionador LDL^T multinivel (parte superior) y del método PCG (parte inferior) en la resolución de (4.1). (a) `ND-HAMD-A` y (b) `ND-HAMD-B`.

Evaluación de los sobrecostes asociados a la extracción del paralelismo de tareas

El último experimento que presentamos en esta sección cuantifica el sobrecoste computacional y de memoria que supone nuestra aproximación para la extracción de paralelismo de tareas en el cálculo del precondicionador multinivel y la resolución iterativa del sistema:

- *Cálculo del precondicionador multinivel.* Según lo especificado en la Sección 2.3, el Algoritmo 2.21 explota el paralelismo de tareas que se pone de manifiesto en las computaciones que realiza el Algoritmo 2.19. El algoritmo paralelo no construye el bloque $\hat{S}_{C_Z}$ (ver (2.103)) de forma explícita, sino que, en su lugar, las tareas calculan y almacenan local e independientemente sus contribuciones a los bloques de $\hat{S}_{C_Z}$, retrasando la acumulación de estas

contribuciones mientras no comience la etapa siguiente (es decir, mientras no sea necesario proceder con la factorización de $\hat{S}_{C_Z}$). Para implementar este principio de funcionamiento, es necesario mantener *bloques locales de contribución* en las estructuras de datos locales de las tareas (consultar el Apartado 2.3.3). Esta decisión permite aumentar el grado de paralelismo durante el cálculo de la factorización parcial (2.103), aunque conlleva un sobrecoste de memoria y computacional. Por una parte, los bloques locales de contribución de dos tareas distintas pueden almacenar *por duplicado* contribuciones a la misma entrada de $\hat{S}_{C_Z}$ (sobrecoste de memoria). Por otra, para calcular dichas contribuciones, el algoritmo paralelo tiene que leer *por duplicado* las entradas de los bloques locales de las tareas, calcular las contribuciones, y escribirlas *por duplicado* en los bloques locales de contribución (sobrecoste computacional).

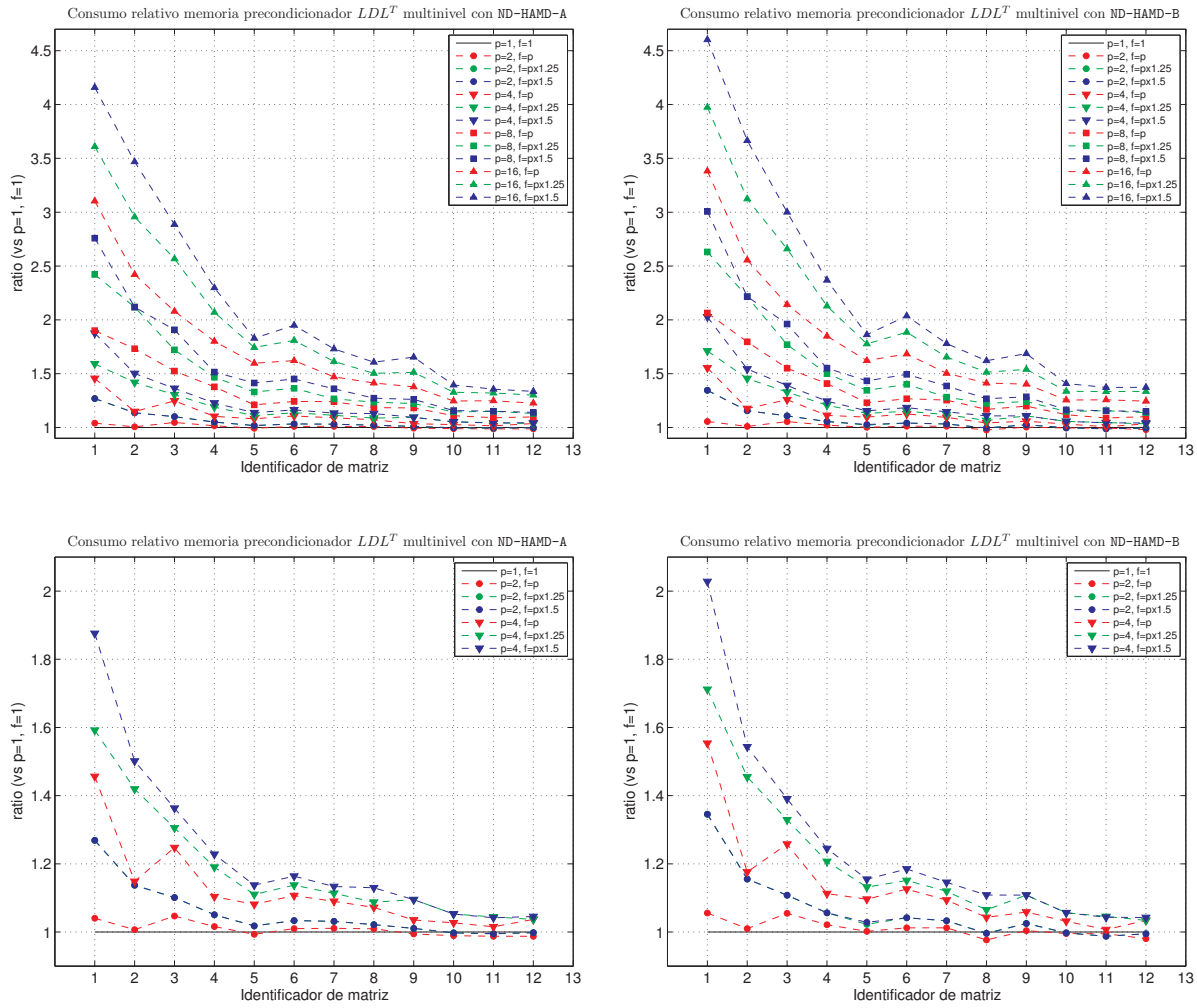
Para evaluar el sobrecoste de memoria, hemos instrumentado los códigos paralelos de tal forma que cada hebra registra, de forma local e independiente y en cada instante de tiempo, la memoria consumida al ejecutar las tareas asignadas por la estrategia de planificación. Durante la ejecución de una tarea, la memoria consumida por el método aumenta al construir su matriz de entrada (ver Figura 2.18), y en el proceso de construcción de una factorización aproximada por bloques local (ver Figuras 2.16 y 2.17), y disminuye al final de cada nivel local cuando se libera la memoria que ocupa el complemento de Schur aproximado del nivel local anterior. Al finalizar la ejecución del algoritmo paralelo, cada hebra determina la *memoria máxima* consumida localmente y, posteriormente, las hebras se sincronizan para calcular la suma de sus máximos locales, obteniendo *la memoria (agregrada) consumida por el algoritmo paralelo*. Es necesario remarcar que la memoria (agregada) consumida por el algoritmo paralelo para construir el precondicionador no coincide con la memoria que ocupa el precondicionador en memoria. Esta última se cuantifica en las Tablas 4.2, 4.3 y 4.4 a través del número de elementos no nulos del precondicionador. En la Figura 4.5 mostramos, para los doce sistemas de la Tabla 4.1 (identificados por el valor correspondiente del eje de abscisas), el *ratio* entre la memoria consumida por el algoritmo secuencial ($p = 1$) y la consumida por el algoritmo paralelo utilizando $p = 2$ (círculos), 4 (triángulos apuntando hacia abajo), 8 (cuadrados), y 16 (triángulos apuntando hacia arriba) procesadores. Así, valores superiores a la unidad en esta figura indican que el consumo agregado de memoria del algoritmo paralelo es superior al del algoritmo secuencial, mientras que valores inferiores a la unidad indican lo contrario. El grado de paralelismo (número de niveles/hojas del árbol) se ajustó mediante el Algoritmo 2.20, variando el valor de f en el intervalo $[p, \frac{3}{2}p]$. En concreto, consideramos $f = p$ (curvas de color rojo), $f = p \times 1,25$ (curvas de color verde) y $f = p \times 1,5$ (curvas de color azul). La parte superior de la Figura 4.5 (a) presenta la comparación cuando se utiliza `ND-HAMD-A` como estrategia para la etapa inicial de particionado, y la parte superior de la Figura 4.5 (b) cuando se utiliza `ND-HAMD-B`; la parte inferior de cada una de estas dos figuras presenta el detalle para el caso de $p = 2$, y $p = 4$ procesadores. Los resultados de las Figura 4.5 (a) y (b) revelan que la memoria consumida por el algoritmo paralelo aumenta con el valor de f. Al aumentar el valor de f, las nuevas tareas hoja que se obtienen almacenan los bloques locales de contribución correspondientes a un mayor número de ancestros, aumentando el número de contribuciones que se almacenan por duplicado y, en consecuencia, el sobrecoste de memoria. El ritmo de crecimiento del sobrecoste de memoria depende de cada matriz, y fue, a partir de la matriz de menor dimensión (código > 1), *sublogarítmico (en base 2) con el número de procesadores*. Este ritmo de crecimiento es moderado teniendo en cuenta que la memoria física disponible en el computadores paralelos crece típicamente de forma lineal con el número de procesadores. Por otra parte, el sobrecoste de memoria disminuye (generalmente) con el tamaño de sistema (para un valor de f fijo). Al aumentar el tamaño de problema, el tamaño relativo de los separadores (es decir, el tamaño relativo de los bloques locales de contribución) disminuye (generalmente) y, en consecuencia, se reduce

el impacto que tiene el almacenamiento por duplicado de las contribuciones locales de las tareas sobre el consumo de memoria agregado del método paralelo.

Para evaluar el sobrecoste computacional, ejecutamos el algoritmo paralelo con un sólo procesador físico, y medimos el tiempo de ejecución para valores de f progresivamente más grandes (en particular, los mismos valores considerados para la evaluación del sobrecoste de memoria). La parte superior de la Figura 4.6 muestra, para los doce sistemas de la Tabla 4.1, el *ratio* entre el tiempo de ejecución del algoritmo paralelo ejecutado con un sólo procesador y el tiempo de ejecución del algoritmo secuencial. Así, valores superiores a la unidad en esta figura indican que el coste agregado del algoritmo paralelo es superior al del algoritmo secuencial, mientras que valores inferiores a la unidad indican lo contrario. La parte superior de la Figura 4.6 (a) presenta la comparación cuando se utiliza `ND-HAMD-A` como estrategia para la etapa inicial de particionado, y la parte superior de la Figura 4.6 (b) cuando se utiliza `ND-HAMD-B`. Los resultados de la parte superior de las Figuras 4.6 (a) y (b) revelan que, para las matrices de menor dimensión (código $= 1, 2, 3, 4$), el coste agregado del algoritmo paralelo crece con respecto al del algoritmo secuencial. Al aumentar el valor de f, las nuevas tareas hoja que se obtienen almacenan los bloques locales de contribución correspondientes a un mayor número de ancestros, aumentando el número de accesos que se realizan por duplicado y, en consecuencia, el sobrecoste computacional. Sin embargo, para las matrices de tamaño intermedio y superior, no siempre se observa esta tendencia. Al aumentar el tamaño del sistema, el peso relativo de este sobrecoste se reduce progresivamente hasta tal punto que ya no domina el coste agregado del algoritmo paralelo, poniéndose de manifiesto otros factores al aumentar el valor de f, como, por ejemplo, la diferente distribución de los nodos entre los niveles algebraicos, el mejor aprovechamiento de la localidad o, en el caso de `ND-HAMD-B`, la reducción del coste agregado asociado a la reordenación de un mayor número de subgrafos independientes más pequeños. Este argumento justifica por qué el coste agregado del algoritmo paralelo es inferior al del algoritmo secuencial en algunos casos (ver, por ejemplo, parte superior de la Figura 4.6 (a) y las tres matrices de mayor dimensión), o por qué, para la tres matrices de mayor dimensión y `ND-HAMD-B` (ver parte superior de la Figura 4.6 (b)), el coste agregado del algoritmo paralelo se reduce al aumentar valor de f.

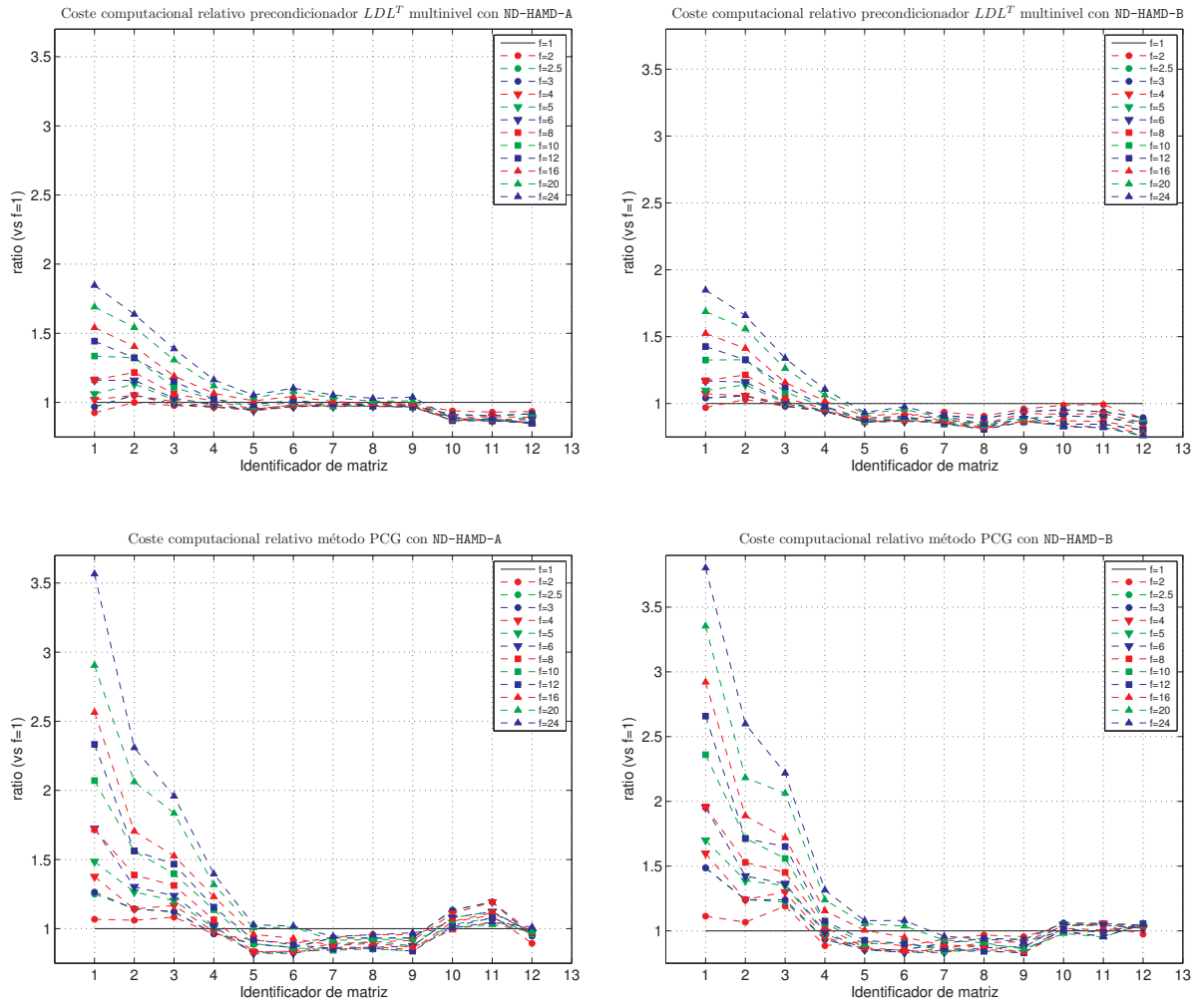
- *Resolución iterativa del sistema.* El método PCG paralelo descompone los vectores $\tilde{b}$, $\tilde{r}$ y $\tilde{q}$ conforme a (3.80), y los vectores $\tilde{x}$, $\tilde{p}$ y $\tilde{z}$ conforme a (3.81). Los bloques locales de las tareas que comparten un ancestro común almacenan *por duplicado* contribuciones a las entradas de los vectores $\tilde{b}$, $\tilde{r}$ y $\tilde{q}$, y réplicas de las entradas de los vectores $\tilde{x}$, $\tilde{p}$ y $\tilde{z}$. El almacenamiento y acceso (escrituras/lecturas) a estas contribuciones/réplicas supone un sobrecoste de memoria y computacional para las operaciones involucradas en la resolución iterativa del sistema. En el caso de los productos escalares, actualizaciones de vectores, y el producto matriz-vector disperso, el sobrecoste computacional también se manifiesta en forma de operaciones aritméticas adicionales. El presente experimento sólo cuantifica el sobrecoste computacional que supone la extracción de paralelismo de tareas en la resolución iterativa del sistema. El consumo de memoria de esta etapa es despreciable en relación al del proceso de construcción del precondicionador y, por tanto, omitimos el estudio del sobrecoste de memoria.

  Para evaluar el sobrecoste computacional asociado a la extracción de paralelismo de tareas en la resolución iterativa del sistema, hemos realizado el mismo experimento que en el caso del cálculo del precondicionador. La parte inferior de la Figura 4.6 ilustra, para los doce sistemas de la Tabla 4.1, el *ratio* entre el tiempo de ejecución del método PCG paralelo ejecutado con un sólo procesador y el tiempo de ejecución del método PCG secuencial. La parte inferior de la Figura 4.6 (a) presenta la comparación cuando se utiliza `ND-HAMD-A` como estrategia para la etapa inicial de particionado, y la parte inferior de la Figura 4.6 (b)



(a) (b)

Figura 4.5: Sobrecoste de memoria que supone aumentar el grado de paralelismo en el cómputo del precondicionador LDL^T multinivel para el problema (4.1) (parte superior) y detalle para el caso de $p = 2$ y $p = 4$ procesadores (parte inferior). (a) `ND-HAMD-A` y (b) `ND-HAMD-B`.



(a) (b)

Figura 4.6: Sobrecoste computacional que supone aumentar el grado de paralelismo en el cómputo del precondicionador LDL^T multinivel (parte superior) y en la resolución iterativa del sistema precondicionado (parte inferior) para el problema (4.1). (a) `ND-HAMD-A` y (b) `ND-HAMD-B`.

cuando se utiliza `ND-HAMD-B`. Los resultados de la parte inferior de las Figuras 4.6 (a) y (b) revelan que, para las matrices de menor dimensión (código $= 1, 2, 3, 4$), el valor del ratio aumenta con f. Al aumentar el valor de f, las nuevas tareas hoja que se obtienen almacenan los bloques locales correspondientes a un mayor número de ancestros, aumentando el número de accesos/operaciones que se realizan por duplicado y, en consecuencia, el sobrecoste computacional. Este sobrecoste afectó en mayor medida al coste agregado del método PCG paralelo que al del precondicionador paralelo (comparar parte inferior y superior de las Figuras 4.6 (a) y (b)). Creemos que esto se debe (principalmente) al mayor grado de reutilización de datos presente en las computaciones involucradas en el precondicionador multinivel, que permite ocultar en mayor grado los accesos duplicados en la caché del procesador. En cualquier caso, al aumentar el tamaño de matriz, el peso relativo de este sobrecoste se reduce progresivamente hasta tal punto que ya no domina el coste agregado del método PCG paralelo. En este punto, la variación del coste agregado del método PCG paralelo en función de f depende de otros factores algorítmicos y arquitecturales causados por las diferencias entre el método secuencial y el paralelo.

4.4. Comparación numérica de los precondicionadores secuencial y paralelo

Nuestro enfoque para la extracción de paralelismo de tareas en el cálculo del precondicionador LDL^T multinivel se basa en una *transformación* del algoritmo básico de ILUPACK; ver Algoritmo 2.19. Esta transformación afecta a la distribución de las incógnitas aceptadas y rechazadas por el método en cada uno de los niveles algebraicos de la jerarquía, por lo que en un principio podría parecer que nuestro enfoque modifica por completo la semántica (es decir, las propiedades numéricas) de las técnicas de precondicionado de la biblioteca. En esta sección proporcionamos evidencias experimentales adicionales en sentido contrario, poniendo de manifiesto la similitud entre la jerarquía multinivel construida por el método secuencial de ILUPACK y la construida por el algoritmo paralelo.

La disección anidada es un heurístico apropiado para la extracción de paralelismo que también resulta útil como estrategia de reordenación para la reducción de llenado. A continuación, esbozamos el comportamiento del enfoque secuencial de precondicionado cuando se aplica a un sistema que ha sido reordenado mediante la disección anidada. En este punto del desarrollo es necesario volver sobre el principio de funcionamiento del núcleo de factorización LDL^T incompleta con pivotamiento basado en la inversa. De acuerdo con el Apartado 2.2.2, en cada paso j de esta factorización, se requiere que

$$\max_{k=j,\ldots,n} \|e_k^T \tilde{L}_j^{-1}\|_\infty \leq \kappa.$$

En general, tenemos que

$$\|e_k^T \tilde{L}_j^{-1}\|_\infty = \max_{\|z\|_\infty=1} |e_k^T \tilde{L}^{-1} z|,$$

y un estimador económico de esta norma consiste en la elección un vector específico $\hat{z}$, con entradas ± 1, de tal forma que $t_k = |e_k^T \tilde{L}^{-1} \hat{z}|$ sea lo más cercano posible a su umbral superior $\max_{\|z\|_\infty=1} |e_k^T \tilde{L}^{-1} z|$; ver Algoritmo 2.11 para más detalles. El estimador $t_k = |y_k|$ se obtiene durante el proceso de factorización resolviendo el sistema triangular inferior $\tilde{L} y = \hat{z}$. Por lo tanto, en el paso j-ésimo de este proceso, el vector y se actualiza conforme a $y_k \leftarrow y_k - \tilde{l}_{kj} y_j$ para aquellas componentes $\tilde{l}_{kj} \neq 0$, con $k > j$. Teniendo en cuenta que el heurístico que calcula t_k determina el valor de las componentes de $\hat{z}$ con el propósito de maximizar $|y_k|$, es probable que la magnitud de aquellas componentes de y que se actualizan con mucha frecuencia acabe siendo "grande". Por construcción de la disección anidada, se espera más llenado en las filas asociadas a los separadores que en las asociadas a los subgrafos independientes. Es decir, cuando se aplica

el enfoque secuencial de precondicionado de ILUPACK a un sistema reordenado con la disección anidada, *es probable que las incógnitas asociadas a los separadores sean rechazadas con mayor frecuencia en los primeros niveles algebraicos de la jerarquía construida por el método numérico.* Los resultados experimentales que recogemos confirman este comportamiento.

El siguiente experimento explora el número y distribución de las incógnitas aceptadas y rechazadas por el método secuencial de ILUPACK cuando se aplica a un sistema reordenado mediante la disección anidada. Con este propósito aplicamos el Algoritmo 2.15 a la reordenación, $A \rightarrow \Omega^T A \Omega$, derivada de la numeración por niveles construida a partir de la estrategias `ND-HAMD-A` o `ND-HAMD-B`, con $l = 2, 3, 4$ o 5 niveles de recursividad. Por ejemplo, en el caso del sistema C, y $l = 3$, el Algoritmo 2.15 se aplicó a las matrices cuyo patrón de dispersidad se muestra en las Figuras 4.3 (b) y (c). En lo que respecta a la configuración del Algoritmo 2.15, hemos desactivado el empleo de heurísticos de reducción de llenado (es decir, $P = I$ en el paso 2 del Algoritmo 2.15) con el propósito de preservar la estructura (patrón de dispersidad) de la matriz de entrada para el primer nivel (es decir, $A \equiv \Omega^T A \Omega$), y la del complemento de Schur aproximado calculado en este nivel y en niveles posteriores. Este ajuste del algoritmo básico se asemeja al Algoritmo 2.19, dado que este último no puede reordenar el bloque A_Z (es decir, la parte del sistema correspondiente a fases posteriores) en el paso 2. Por último, hemos escogido $\kappa = 5$ y $\tau = 10^{-2}$ para el núcleo de factorización con pivotamiento basado en la inversa (ver paso 3 del Algoritmo 2.15).

Las Tablas 4.5, 4.6 y 4.7 detallan la distribución de cada uno de los niveles algebraicos construidos por el Algoritmo 2.15 con respecto a los niveles de recursividad de la disección anidada (niveles del árbol). Las Tablas 4.5 y 4.6 muestran esta distribución para los sistemas F y VF1, respectivamente, y todas las combinaciones de estrategia y número de niveles de recursividad de la disección anidada, mientras que la Tabla 4.7 se centra en los sistemas M, VC2, M1, VF, F1, C2, VC3 y M2, y una sola combinación de estrategia y número de niveles de recursividad de la disección anidada (en particular, `ND-HAMD-A` y $l = 5$, respectivamente). Para cada combinación de sistema, estrategia y valor de l en las tablas, las columnas etiquetadas como "Niv. prec." y "Tamaño Total" muestran, respectivamente, el identificador de nivel dentro de la jerarquía multinivel construida por el método numérico, y el tamaño de la matriz de entrada para dicho nivel (es decir, el tamaño de la matriz A en el paso 1 del Algoritmo 2.15). Por lo tanto, el tamaño de la matriz de entrada para el primer nivel (Niv. prec. = 1) siempre coincide con el tamaño de la matriz del sistema; ver Tabla 4.1. Además, el número de incógnitas aceptadas en el nivel i-ésimo se obtiene restando el tamaño de las matrices de entrada para los niveles i-ésimo e $i+1$-ésimo, y el de incógnitas rechazadas coincide con el tamaño de la matriz de entrada para el nivel $i+1$-ésimo. Por ejemplo, en la Tabla 4.5, `ND-HAMD-A` y $l = 3$, el número de incógnitas aceptadas por el método en el segundo nivel (Niv. prec. = 2) es 18.052 (27.343-9.291) y, el de rechazadas, 9.291. La distribución de cada uno de los niveles algebraicos con respecto a los niveles de recursividad de la disección anidada (niveles del árbol) se muestra en las columnas etiquetadas como "Nivel árbol". Estas columnas descomponen las incógnitas de la matriz de entrada para cada nivel algebraico en función del nivel de recursividad de la disección anidada al que pertenecen. Para una combinación de nivel algebraico (valor de la columna "Niv. prec.") y nivel de recursividad (columna situada debajo de la etiqueta "Nivel árbol"), la celda correspondiente indica el número de incógnitas de la matriz de entrada para dicho nivel algebraico que pertenecen a dicho nivel de recursividad y, entre paréntesis, el *ratio* entre el número de incógnitas que faltan por procesar de ese nivel de recursividad a la entrada de dicho nivel algebraico y el número de incógnitas que pertenecen a dicho nivel de recursividad en el primer nivel algebraico. Por ejemplo, en la Tabla 4.6, `ND-HAMD-B` y $l = 3$, 5.367.030 incógnitas pertenecen a los subgrafos independientes $G_{(3,1)}$, $G_{(3,2)}$, $G_{(3,3)}$ y $G_{(3,4)}$ (tercer nivel de la recursividad), 23.101 a los separadores $S_{(2,1)}$ y $S_{(2,2)}$ (segundo nivel de la recursividad), y 23.389 al separador $S_{(1,1)}$ (primer nivel de la recursividad). A la entrada del tercer nivel algebraico (Niv. prec. = 3), faltan por procesar 853.853 incógnitas (16 %) de los subgrafos independientes,

17.565 (76 %) de los separadores del segundo nivel, y 18.201 (78 %) del separador del primer nivel.

Cód.	l	Opc. ND-HAMD-X	Niv. prec.	Tamaño Total	Nivel árbol 1	2	3	4	5
F	2	A	1	101.296	1.653 (1,00)	99.643 (1,00)			
F	2	A	2	27.642	1.557 (0,94)	26.085 (0,26)			
F	2	A	3	9.031	1.368 (0,83)	7.663 (0,08)			
F	2	A	4	2.175	751 (0,45)	1.424 (0,01)			
F	2	A	5	213	116 (0,07)	97 (0,00)			
F	2	B	1	101.296	1.653 (1,00)	99.643 (1,00)			
F	2	B	2	27.343	1.519 (0,92)	25.824 (0,26)			
F	2	B	3	9.291	1.171 (0,71)	8.120 (0,08)			
F	2	B	4	2.132	773 (0,47)	1.359 (0,01)			
F	2	B	5	64	49 (0,03)	15 (0,00)			
F	3	A	1	101.296	1.653 (1,00)	1.466 (1,00)	98.177 (1,00)		
F	3	A	2	27.642	1.557 (0,94)	1.369 (0,93)	24.716 (0,25)		
F	3	A	3	8.981	1.323 (0,80)	1.089 (0,74)	6.569 (0,07)		
F	3	A	4	1.948	599 (0,36)	534 (0,36)	815 (0,01)		
F	3	A	5	143	61 (0,04)	76 (0,05)	6 (0,00)		
F	3	B	1	101.296	1.653 (1,00)	1.466 (1,00)	98.177 (1,00)		
F	3	B	2	27.385	1.538 (0,93)	1.327 (0,91)	24.520 (0,25)		
F	3	B	3	8.824	1.154 (0,70)	1.019 (0,70)	6.651 (0,07)		
F	3	B	4	2.002	561 (0,34)	430 (0,29)	1.011 (0,01)		
F	3	B	5	166	111 (0,07)	4 (0,00)	51 (0,00)		
F	4	A	1	101.296	1.653 (1,00)	1.466 (1,00)	1.455 (1,00)	96.722 (1,00)	
F	4	A	2	27.642	1.557 (0,94)	1.369 (0,93)	1.332 (0,92)	23.384 (0,24)	
F	4	A	3	8.990	1.316 (0,80)	1.070 (0,73)	1.018 (0,70)	5.586 (0,06)	
F	4	A	4	2.034	567 (0,34)	548 (0,37)	480 (0,33)	439 (0,00)	
F	4	A	5	133	55 (0,03)	73 (0,05)	5 (0,00)	0 (0,00)	
F	4	B	1	101.296	1.653 (1,00)	1.466 (1,00)	1.455 (1,00)	96.722 (1,00)	
F	4	B	2	27.443	1.549 (0,94)	1.334 (0,91)	1.332 (0,92)	23.228 (0,24)	
F	4	B	3	9.364	1.237 (0,75)	1.074 (0,73)	1.134 (0,78)	5.919 (0,06)	
F	4	B	4	1.872	403 (0,24)	419 (0,29)	589 (0,40)	461 (0,00)	
F	4	B	5	13	11 (0,01)	0 (0,00)	2 (0,00)	0 (0,00)	
F	5	A	1	101.296	1.653 (1,00)	1.466 (1,00)	1.455 (1,00)	2.204 (1,00)	94.518 (1,00)
F	5	A	2	27.651	1.557 (0,94)	1.371 (0,94)	1.336 (0,92)	2.025 (0,92)	21.362 (0,23)
F	5	A	3	9.037	1.302 (0,79)	1.077 (0,73)	1.004 (0,69)	1.572 (0,71)	4.082 (0,04)
F	5	A	4	2.009	610 (0,37)	538 (0,37)	439 (0,30)	418 (0,19)	4 (0,00)
F	5	A	5	79	11 (0,01)	68 (0,05)	0 (0,00)	0 (0,00)	0 (0,00)
F	5	B	1	101.296	1.653 (1,00)	1.466 (1,00)	1.455 (1,00)	2.204 (1,00)	94.518 (1,00)
F	5	B	2	27.366	1.532 (0,93)	1.313 (0,90)	1.335 (0,92)	1.986 (0,90)	21.200 (0,22)
F	5	B	3	9.030	1.320 (0,80)	1.065 (0,73)	1.080 (0,74)	1.650 (0,75)	3.915 (0,04)
F	5	B	4	2.040	859 (0,52)	544 (0,37)	406 (0,28)	230 (0,10)	1 (0,00)
F	5	B	5	115	115 (0,07)	0 (0,00)	0 (0,00)	0 (0,00)	0 (0,00)

Tabla 4.5: Distribución de los niveles algebraicos construidos por el Algoritmo 2.15 entre los niveles de recursividad de la disección anidada (niveles del árbol) en el caso del sistema F; ver Tabla 4.1. El algoritmo se aplicó sobre la reordenación de F derivada de la numeración por niveles construida a partir de las estrategias `ND-HAMD-A` o `ND-HAMD-B` con $l = 2, 3, 4$, o $l = 5$ niveles de recursividad.

Los resultados de las Tablas 4.5, 4.6 y 4.7 evidencian una tendencia muy marcada de la distribución de las incógnitas aceptadas y rechazadas por el método numérico. A continuación, ilustramos dicha tendencia para el caso del sistema VF1, `ND-HAMD-B` y $l = 3$ en la Tabla 4.6. En el primer nivel algebraico, el método *sólo* rechaza el 30 % de las incógnitas que pertenecen a los subgrafos independientes. Sin embargo, rechaza el 89 % y el 90 % de las incógnitas asociadas a los separadores del segundo nivel y del primer nivel, respectivamente. En el segundo nivel algebraico, el método rechaza el 51 % $(0,16/0,31)$ de las incógnitas de los subgrafos independientes, mientras

Cód.	l	Opc. ND-HAMD-X	Niv. prec.	Tamaño Total	Nivel árbol 1	2	3	4	5
VF1	2	A	1	5.413.520	23.389 (1,00)	5.390.131 (1,00)			
VF1	2	A	2	1.771.934	21.175 (0,91)	1.750.759 (0,32)			
VF1	2	A	3	919.850	18.670 (0,80)	901.180 (0,17)			
VF1	2	A	4	462.598	14.324 (0,61)	448.274 (0,08)			
VF1	2	A	5	151.556	7.092 (0,30)	144.464 (0,03)			
VF1	2	A	6	10.637	972 (0,04)	9.665 (0,00)			
VF1	2	A	7	4	0 (0,00)	4 (0,00)			
VF1	2	B	1	5.413.520	23.389 (1,00)	5.390.131 (1,00)			
VF1	2	B	2	1.702.301	20.920 (0,89)	1.681.381 (0,31)			
VF1	2	B	3	884.692	18.257 (0,78)	866.435 (0,16)			
VF1	2	B	4	464.056	14.862 (0,64)	449.194 (0,08)			
VF1	2	B	5	189.858	9.654 (0,41)	180.204 (0,03)			
VF1	2	B	6	30.951	3.305 (0,14)	27.646 (0,01)			
VF1	2	B	7	208	15 (0,00)	193 (0,00)			
VF1	3	A	1	5.413.520	23.389 (1,00)	23.101 (1,00)	5.367.030 (1,00)		
VF1	3	A	2	1.771.939	21.176 (0,91)	20.950 (0,91)	1.729.813 (0,32)		
VF1	3	A	3	919.413	18.626 (0,80)	18.206 (0,79)	882.581 (0,16)		
VF1	3	A	4	462.005	14.094 (0,60)	13.536 (0,59)	434.375 (0,08)		
VF1	3	A	5	153.032	7.593 (0,32)	7.086 (0,31)	138.353 (0,03)		
VF1	3	A	6	10.133	705 (0,03)	814 (0,04)	8.614 (0,00)		
VF1	3	B	1	5.413.520	23.389 (1,00)	23.101 (1,00)	5.367.030 (1,00)		
VF1	3	B	2	1.704.384	20.966 (0,90)	20.581 (0,89)	1.662.837 (0,31)		
VF1	3	B	3	889.619	18.201 (0,78)	17.565 (0,76)	853.853 (0,16)		
VF1	3	B	4	463.572	14.487 (0,62)	13.871 (0,60)	435.214 (0,08)		
VF1	3	B	5	190.927	8.739 (0,37)	8.657 (0,37)	173.531 (0,03)		
VF1	3	B	6	31.366	2.472 (0,11)	2.650 (0,11)	26.244 (0,00)		
VF1	3	B	7	105	10 (0,00)	27 (0,00)	68 (0,00)		
VF1	4	A	1	5.413.520	23.389 (1,00)	23.101 (1,00)	22.508 (1,00)	5.344.522 (1,00)	
VF1	4	A	2	1.771.939	21.176 (0,91)	20.950 (0,91)	20.525 (0,91)	1.709.288 (0,32)	
VF1	4	A	3	919.374	18.772 (0,80)	18.041 (0,78)	18.306 (0,81)	864.255 (0,16)	
VF1	4	A	4	463.003	14.186 (0,61)	13.268 (0,57)	13.534 (0,60)	422.015 (0,08)	
VF1	4	A	5	153.816	7.097 (0,30)	6.538 (0,28)	7.339 (0,33)	132.842 (0,02)	
VF1	4	A	6	10.567	961 (0,04)	918 (0,04)	929 (0,04)	7.759 (0,00)	
VF1	4	A	7	34	0 (0,00)	32 (0,00)	1 (0,00)	1 (0,00)	
VF1	4	B	1	5.413.520	23.389 (1,00)	23.101 (1,00)	22.508 (1,00)	5.344.522 (1,00)	
VF1	4	B	2	1.700.983	20.777 (0,89)	20.655 (0,89)	19.926 (0,89)	1.639.625 (0,31)	
VF1	4	B	3	881.731	18.143 (0,78)	17.991 (0,78)	17.083 (0,76)	828.514 (0,16)	
VF1	4	B	4	457.488	14.478 (0,62)	13.434 (0,58)	13.566 (0,60)	416.010 (0,08)	
VF1	4	B	5	180.161	9.419 (0,40)	8.132 (0,35)	7.197 (0,32)	155.413 (0,03)	
VF1	4	B	6	27.831	2.958 (0,13)	2.021 (0,09)	2.032 (0,09)	20.820 (0,00)	
VF1	4	B	7	310	85 (0,00)	31 (0,00)	74 (0,00)	120 (0,00)	
VF1	5	A	1	5.413.520	23.389 (1,00)	23.101 (1,00)	22.508 (1,00)	30.653 (1,00)	5.313.869 (1,00)
VF1	5	A	2	1.771.929	21.166 (0,90)	20.954 (0,91)	20.525 (0,91)	27.819 (0,91)	1.681.465 (0,32)
VF1	5	A	3	920.023	18.712 (0,80)	18.231 (0,79)	18.252 (0,81)	24.561 (0,80)	840.267 (0,16)
VF1	5	A	4	461.831	13.857 (0,59)	13.212 (0,57)	13.834 (0,61)	17.306 (0,56)	403.622 (0,08)
VF1	5	A	5	150.741	7.391 (0,32)	6.277 (0,27)	7.144 (0,32)	9.418 (0,31)	120.511 (0,02)
VF1	5	A	6	9.377	993 (0,04)	422 (0,02)	736 (0,03)	1.038 (0,03)	6.188 (0,00)
VF1	5	A	7	1	0 (0,00)	0 (0,00)	0 (0,00)	0 (0,00)	1 (0,00)
VF1	5	B	1	5.413.520	23.389 (1,00)	23.101 (1,00)	22.508 (1,00)	30.653 (1,00)	5.313.869 (1,00)
VF1	5	B	2	1.705.277	20.840 (0,89)	20.704 (0,90)	20.182 (0,90)	27.489 (0,90)	1.616.062 (0,30)
VF1	5	B	3	891.310	18.343 (0,78)	18.231 (0,79)	17.609 (0,78)	24.054 (0,78)	813.073 (0,15)
VF1	5	B	4	467.650	14.571 (0,62)	14.562 (0,63)	13.629 (0,61)	18.319 (0,60)	406.569 (0,08)
VF1	5	B	5	187.645	9.185 (0,39)	8.898 (0,39)	8.343 (0,37)	10.275 (0,34)	150.944 (0,03)
VF1	5	B	6	29.144	2.501 (0,11)	2.684 (0,12)	1.903 (0,08)	2.749 (0,09)	19.307 (0,00)
VF1	5	B	7	284	82 (0,00)	60 (0,00)	44 (0,00)	69 (0,00)	29 (0,00)

Tabla 4.6: Distribución de los niveles algebraicos construidos por el Algoritmo 2.15 entre los niveles de recursividad de la disección anidada (niveles del árbol) en el caso del sistema VF1; ver Tabla 4.1. El algoritmo se aplicó sobre la reordenación de VF1 derivada de la numeración por niveles construida a partir de las estrategias `ND-HAMD-A` o `ND-HAMD-B` con $l = 2, 3, 4$, o $l = 5$ niveles de recursividad.

que rechaza *de nuevo* el 85 % (0, 76/0, 89) y el 86 % (0, 78/0, 90) de las incógnitas pertenecientes a los separadores del segundo y del primer nivel. *Esta tendencia continúa hasta que la mayor parte de los nodos de los subgrafos independientes han sido aceptados, y sólo a partir de este momento el método empieza a aceptar la mayoría de los nodos pertenecientes a los separadores.*

Cód.	l	Opc. ND-HAMD-X	Niv. prec.	Tamaño Total	Nivel árbol 1	2	3	4	5
M	5	A	1	32.429	703 (1,00)	656 (1,00)	690 (1,00)	740 (1,00)	29.640 (1,00)
M	5	A	2	6.808	633 (0,90)	608 (0,93)	605 (0,88)	674 (0,91)	4.288 (0,14)
M	5	A	3	1.377	445 (0,63)	441 (0,67)	375 (0,54)	116 (0,16)	0 (0,00)
VC2	5	A	1	271.272	2.948 (1,00)	2.635 (1,00)	3.002 (1,00)	3.885 (1,00)	258.802 (1,00)
VC2	5	A	2	70.088	2.501 (0,85)	2.206 (0,84)	2.508 (0,84)	3.284 (0,85)	59.589 (0,23)
VC2	5	A	3	27.120	1.863 (0,63)	1.679 (0,64)	2.028 (0,68)	2.737 (0,70)	18.813 (0,07)
VC2	5	A	4	4.704	693 (0,24)	764 (0,29)	932 (0,31)	1.114 (0,29)	1.201 (0,00)
M1	5	A	1	297.927	3.347 (1,00)	3.227 (1,00)	3.235 (1,00)	3.173 (1,00)	284.945 (1,00)
M1	5	A	2	84.003	2.945 (0,88)	2.854 (0,88)	2.824 (0,87)	2.867 (0,90)	72.513 (0,25)
M1	5	A	3	34.576	2.543 (0,76)	2.372 (0,74)	2.380 (0,74)	2.460 (0,78)	24.821 (0,09)
M1	5	A	4	6.830	1.431 (0,43)	1.278 (0,40)	1.071 (0,33)	1.015 (0,32)	2.035 (0,01)
M1	5	A	5	734	342 (0,10)	235 (0,07)	155 (0,05)	1 (0,00)	1 (0,00)
VF	5	A	1	658.609	6.264 (1,00)	5.021 (1,00)	5.624 (1,00)	6.763 (1,00)	634.937 (1,00)
VF	5	A	2	204.335	5.984 (0,96)	4.743 (0,94)	5.311 (0,94)	6.330 (0,94)	181.967 (0,29)
VF	5	A	3	92.757	5.353 (0,85)	4.271 (0,85)	4.679 (0,83)	5.702 (0,84)	72.752 (0,11)
VF	5	A	4	32.440	3.639 (0,58)	3.180 (0,63)	3.485 (0,62)	3.767 (0,56)	18.369 (0,03)
VF	5	A	5	4.307	993 (0,16)	780 (0,16)	796 (0,14)	1152 (0,17)	586 (0,00)
VF	5	A	6	250	179 (0,03)	0 (0,00)	71 (0,01)	0 (0,00)	0 (0,00)
F1	5	A	1	882.824	6.844 (1,00)	6.469 (1,00)	5.902 (1,00)	8.502 (1,00)	855.107 (1,00)
F1	5	A	2	264.278	6.047 (0,88)	5.747 (0,89)	5.292 (0,90)	7.701 (0,91)	239.491 (0,28)
F1	5	A	3	117.919	5.428 (0,79)	5.091 (0,79)	4.555 (0,77)	6.680 (0,79)	96.165 (0,11)
F1	5	A	4	41.826	3.796 (0,55)	3.425 (0,53)	3.272 (0,55)	4.737 (0,56)	26.596 (0,03)
F1	5	A	5	4.940	1.419 (0,21)	727 (0,11)	669 (0,11)	1.226 (0,14)	899 (0,00)
C2	5	A	1	906.882	6.846 (1,00)	6.207 (1,00)	7.362 (1,00)	9.265 (1,00)	877.202 (1,00)
C2	5	A	2	264.101	6.093 (0,89)	5.438 (0,88)	6.329 (0,86)	8.164 (0,88)	238.077 (0,27)
C2	5	A	3	110.960	5.264 (0,77)	4.462 (0,72)	5.335 (0,72)	6.585 (0,71)	89.314 (0,10)
C2	5	A	4	35.100	3.432 (0,50)	3.306 (0,53)	3.624 (0,49)	4.051 (0,44)	20.687 (0,02)
C2	5	A	5	3.256	1.116 (0,16)	826 (0,13)	555 (0,08)	556 (0,06)	203 (0,00)
VC3	5	A	1	2.382.864	13.752 (1,00)	11.687 (1,00)	11.879 (1,00)	15.696 (1,00)	2.329.850 (1,00)
VC3	5	A	2	682.833	11.562 (0,84)	9.894 (0,85)	9.969 (0,84)	13.200 (0,84)	638.208 (0,27)
VC3	5	A	3	288.183	9.583 (0,70)	8.258 (0,71)	8.288 (0,70)	10.702 (0,68)	251.352 (0,11)
VC3	5	A	4	102.036	6.082 (0,44)	5.182 (0,44)	5.072 (0,43)	7.031 (0,45)	78.669 (0,03)
VC3	5	A	5	5.899	1.130 (0,08)	1.111 (0,10)	973 (0,08)	1.373 (0,09)	1.312 (0,00)
VC3	5	A	6	1	1 (0,00)	0 (0,00)	0 (0,00)	0 (0,00)	0 (0,00)
M2	5	A	1	2.539.954	14.249 (1,00)	10.276 (1,00)	12.788 (1,00)	15.809 (1,00)	2.486.832 (1,00)
M2	5	A	2	790.996	12.662 (0,89)	9.286 (0,90)	11.343 (0,89)	14.024 (0,89)	743.681 (0,30)
M2	5	A	3	366.122	10.834 (0,76)	7.923 (0,77)	9.702 (0,76)	11.674 (0,74)	325.989 (0,13)
M2	5	A	4	139.849	6.964 (0,49)	5.450 (0,53)	6.406 (0,50)	7.346 (0,46)	113.683 (0,05)
M2	5	A	5	13.913	2.107 (0,15)	1.300 (0,13)	1.566 (0,12)	1.982 (0,13)	6.958 (0,00)
M2	5	A	6	1	0 (0,00)	1 (0,00)	0 (0,00)	0 (0,00)	0 (0,00)

Tabla 4.7: Distribución de los niveles algebraicos construidos por el Algoritmo 2.15 entre los niveles de recursividad de la disección anidada (niveles del árbol) en el caso de los sistemas M, VC2, M1, VF, F1, C2, VC3, y M2; ver Tabla 4.1. El algoritmo se aplicó sobre la reordenación de estas matrices derivada de la numeración por niveles construida a partir de la estrategia ND-HAMD-A con $l = 5$ niveles de recursividad.

Puede apreciarse la tendencia para el resto de combinaciones de matriz, estrategia y valor de l en las Tablas 4.5, 4.6 y 4.7.

La tendencia observada en las Tablas 4.5, 4.6 y 4.7 es habitual. Al repetir el experimento con diversas matrices de prueba (distintas a las de nuestro problema de aplicación), hemos observado que el algoritmo secuencial no empieza a aceptar la gran mayoría de las incógnitas pertenecientes a los separadores hasta haber eliminado casi la totalidad de la incógnitas asociadas a los subgrafos independientes. Desde este punto de vista, el algoritmo secuencial desaprovecha tiempo de computación rechazando las incógnitas pertenecientes a los separadores, incógnitas que son rechazadas (sin coste alguno) por el principio de funcionamiento del algoritmo paralelo.

La Figura 4.7 compara el número y distribución de las incógnitas aceptadas (puntos azules) y rechazadas (puntos rojos) en los primeros cuatro niveles del Algoritmo 2.15 (izquierda) y del Algoritmo 2.19 (derecha) cuando se aplican al sistema de ecuaciones que surge de la discretización, mediante una malla de 200×200 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (c). Como se puede observar en la Figura 4.7, la distribución de las incógnitas aceptadas y rechazadas es casi idéntica para ambos algoritmos en el caso de las incógnitas pertenecientes a los subgrafos independientes. A pesar de que sí se pueden observar diferencias en esta distribución para el caso de las incógnitas pertenecientes a los separadores, especialmente a partir del tercer nivel algebraico (tercera fila de la Figura 4.7), la figura también revela, para esta sistema de prueba, la tendencia mencionada previamente.

4.5. Evaluación de aproximaciones paralelas para el particionado de matrices dispersas

Los resultados experimentales de la Sección 4.3 (Tablas 4.2, 4.3 y 4.4) revelan que el uso de la estrategia `ND-HAMD-B` reduce el *tiempo de ejecución* de todo el proceso en relación al de `ND-HAMD-A` (por ejemplo, alrededor de un factor 2 con $p = 16$ procesadores). Esta reducción está causada (principalmente) por los dos factores siguientes. En primer lugar, el aprovechamiento del paralelismo de tareas para reordenar los subgrafos independientes durante el cálculo del precondicionador LDL^T multinivel; en segundo lugar, el empleo de HAMD para reordenar los subgrafos independientes, un heurístico más económico que MLND en términos computacionales. Sin embargo, el rápido crecimiento del coste de `ND-HAMD-B` con p (es decir, el coste asociado al cálculo de los separadores de vértices de los primeros niveles de la disección anidada) limita el margen de mejora de esta estrategia (es decir, el volumen de paralelismo que se puede explotar reordenando en paralelo los subgrafos independientes). En esta sección abordamos esta limitación aprovechando el paralelismo para calcular los separadores de vértices. Para ello, nos apoyamos sobre dos paquetes de software paralelos que incorporan los últimos avances en el campo: ParMETIS [105] y PT-SCOTCH [129]. Además de evaluar comparativamente el consumo de memoria y la eficiencia paralela de estos paquetes para particionar los sistemas ligados a la EDP tridimensional (4.1), en la sección también estudiamos el impacto que tiene el particionado calculado sobre las prestaciones de las etapas numéricas de nuestro resolutor.

4.5.1. Revisión de los principios de funcionamiento de ParMETIS y PT-SCOTCH

El enfoque multinivel para el cálculo de separadores de vértices

La mayoría de heurísticos disponibles en la actualidad para el cálculo de separadores de vértices se fundamentan en el *enfoque multinivel* [89, 102]. Esta aproximación combina las potencialidades de los heurísticos de optimización local (rapidez) y de particionado global (calidad) para el cálculo rápido de separadores de vértices de alta calidad. El proceso que realizan los heurísticos multinivel se puede esquematizar en las siguientes tres fases (ver Figura 4.8):

1. "Engrosamiento" (*coarsening phase*). Esta fase consiste en la aplicación *recursiva* de una operación básica denominada "engrosamiento de grafos" (*graph coarsening*). Esta operación recibe como entrada un grafo y devuelve como salida un grafo de menor tamaño, pero con una estructura similar al primero. Este último grafo se denomina grafo "grueso" (*coarse graph*) y, el primero, grafo "fino" (*fine graph*). El grafo grueso se obtiene típicamente calculando una correspondencia (*matching*) de su conjunto de aristas. Una correspondencia, M, es un subconjunto del conjunto de aristas del grafo fino tal que no hay dos aristas en M que compartan un vértice. Los nodos del grafo grueso se obtienen uniendo parejas de nodos del grafo fino conectados a través de una arista de M; los nodos del grafo fino que no estén conectados con otro a través de una arista de M forman parte del grafo grueso

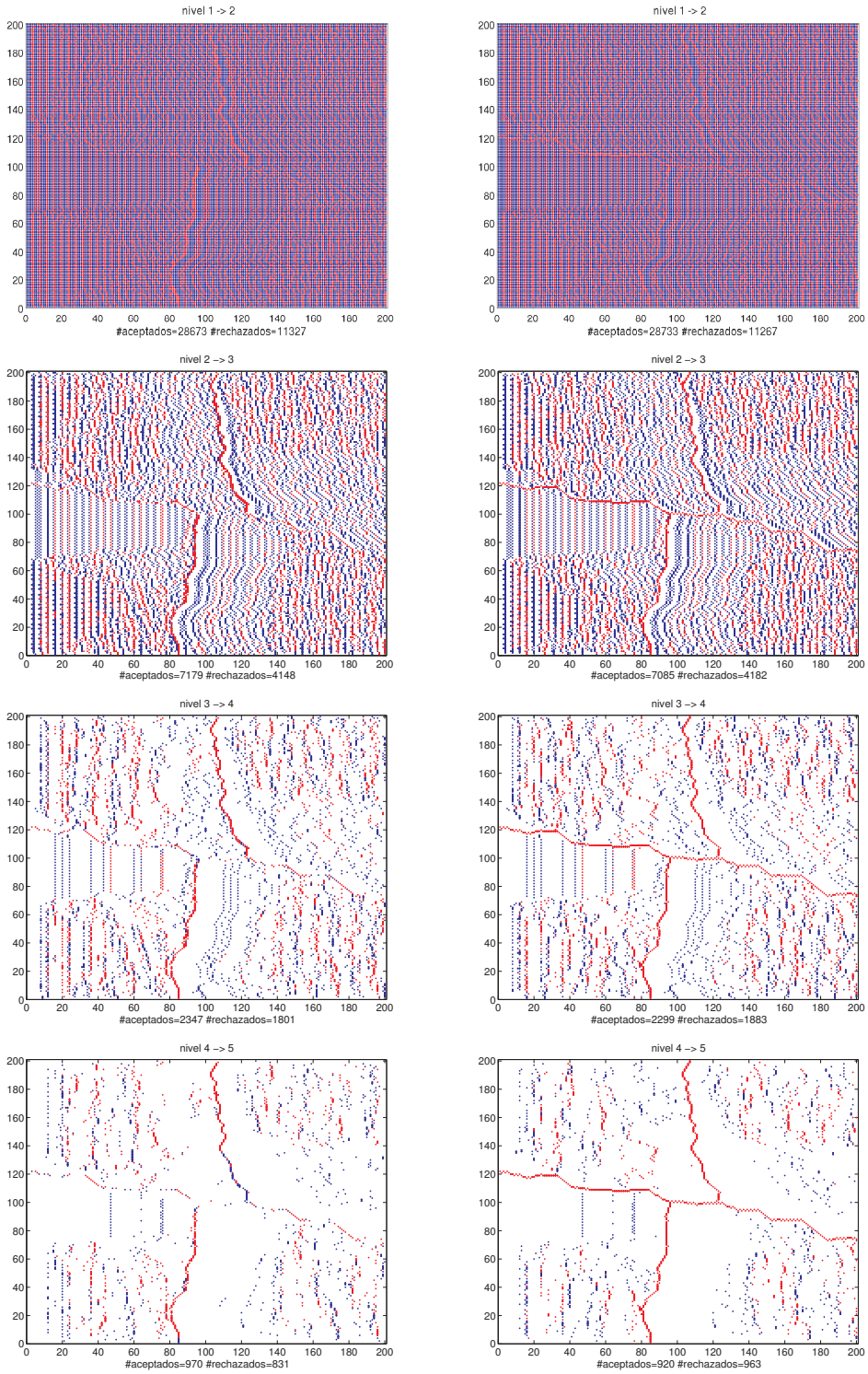


Figura 4.7: Número y distribución de las incógnitas aceptadas (puntos azules) y rechazadas (puntos rojos) en los primeros cuatro niveles del Algoritmo 2.15 (izquierda) y del Algoritmo 2.19 (derecha) cuando se aplican al sistema de ecuaciones que surge de la discretización, mediante una malla de 200×200 nodos, de la variante de la EDP de difusión correspondiente a la Figura 2.20 (c).

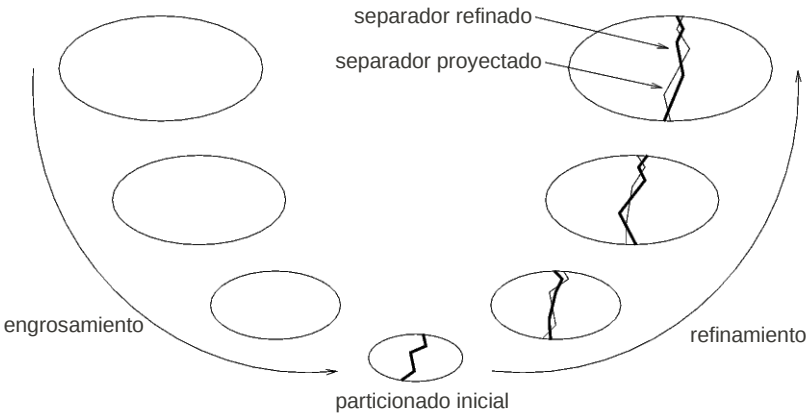


Figura 4.8: El proceso de los heurísticos multinivel para el cálculo de separadores de vértices se descompone en tres fases: engrosamiento (*coarsening phase*), particionado inicial (*initial partitioning stage*) y refinamiento (*uncoarsening phase*).

directamente. Como el objetivo del engrosamiento es reducir el tamaño del grafo fino, M debería contener el *máximo* número (posible) de aristas del grafo fino. Por razones de eficiencia computacional, los heurísticos disponibles (por ejemplo, HEM, LEM, RM, HCM) relajan típicamente este requisito, calculando, en su lugar, correspondencias *maximales* (*maximal matchings*) [102].

La fase de engrosamiento parte de un grafo inicial (por ejemplo, el grafo de adyacencia de la matriz del sistema) y construye a partir de éste una jerarquía de grafos progresivamente más gruesos. En un determinado nivel de esta jerarquía, el algoritmo recibe el grafo del nivel anterior (*finer graph*) y construye un grafo más grueso (*coarser graph*) aplicando la operación descrita en el párrafo anterior. Este proceso recursivo termina típicamente cuando el tamaño del grafo grueso construido en un determinado nivel es inferior a un umbral, aunque también se pueden utilizar otros criterios basados en la calidad de M como, por ejemplo, que el *ratio* entre los tamaños de los grafos correspondientes a dos niveles consecutivos sea "grande" (cuanto mayor sea el valor de este ratio más baja es la calidad de M).

2. Particionado inicial (*initial partitioning stage*). Esta fase del algoritmo calcula un separador de vértices del grafo más grueso de la jerarquía, utilizando para ello heurísticos de particionado global, como por ejemplo, BS, GCP o GGGP.

3. "Refinamiento" (*uncoarsening phase*). Esta fase recibe como entradas el separador de vértices del grafo más grueso y la jerarquía de grafos, y obtiene el separador del grafo más fino (es decir, el resultado del algoritmo). Para ello, partiendo del primer separador, aplica una operación denominada proyección y refinamiento en cada uno de los niveles de la jerarquía de grafos hasta llegar al grafo más fino. En cada nivel de esta jerarquía, esta operación proyecta el separador del grafo inmediatamente más grueso (es decir, el del nivel previo) sobre el grafo del nivel actual. A continuación, lo refina (es decir, mejora su calidad) mediante heurísticos de optimización local, como KL [106] o FM [69].

El consumo de memoria de este tipo de heurísticos está concentrado en la primera fase, dado que es necesario reservar memoria para cada uno de los grafos de la jerarquía. La distribución del coste computacional del heurístico entre estas tres fases depende de cada problema, aunque típicamente la primera y tercera fase concentran casi la totalidad del coste computacional. En lo que respecta a su paralelización, la parte más compleja y desafiante es la fase 3, debido a la naturaleza inherentemente secuencial de los heurísticos de optimización local [104].

Paralelización. ParMETIS y PT-SCOTCH

ParMETIS y PT-SCOTCH son dos paquetes de software paralelos cuyo objetivo fundamental es el particionado de grafos y la reordenación de matrices dispersas, incorporando los avances más recientes en heurísticos multinivel paralelos para el cálculo de separadores de vértices [48, 104]. De hecho, paquetes de software muy asentados en la comunidad científica como PETSc (consultar el Apartado 1.3.3), proporcionan interfaces para acceder a las rutinas de estos paquetes de particionado. La arquitectura destino de ParMETIS y PT-SCOTCH son los multiprocesadores con memoria distribuida, ambos están escritos en el lenguaje C, y utilizan la biblioteca MPI para desarrollar sus implementaciones bajo el modelo de programación paralelo basado en el paso de mensajes. A continuación, presentamos brevemente los tipos de paralelismo que ambos explotan para el proceso de la disección anidada, centrándonos en sus diferencias. Estas diferencias formarán la base para poder justificar los resultados experimentales recogidos en el Apartado 4.5.2.

ParMETIS y PT-SCOTCH explotan tres tipos de paralelismo (aunque utilizando algoritmos y técnicas paralelas diferentes) durante el proceso de la disección anidada multinivel:

- Paralelismo del tipo disección anidada. Es el tipo de paralelismo disponible naturalmente durante el proceso. Los separadores de los subgrafos independientes pertenecientes al mismo nivel de recursividad de la disección anidada pueden ser calculados de forma independiente (es decir, en paralelo) por subconjuntos disjuntos de procesadores. Por ejemplo, apoyémonos en la Figura 4.2, y asumamos que el grafo de adyacencia de la matriz del sistema está distribuido entre p procesadores. Los p procesadores participan en el cálculo paralelo del separador $S_{(1,1)}$. Después de calcular este separador, el subgrafo independiente $G_{(2,1)}$ se puede redistribuir entre una primera mitad de $p/2$ procesadores, y el subgrafo independiente $G_{(2,2)}$, entre otra mitad de $p/2$ procesadores. A partir de este momento, dos subconjuntos disjuntos, de $p/2$ procesadores cada uno, participan de forma totalmente independiente en el cálculo de los separadores $S_{(2,1)}$ y $S_{(2,2)}$.

- Paralelismo del tipo engrosamiento. Es el tipo de paralelismo que se puede explotar durante la fase de engrosamiento mediante el empleo de algoritmos paralelos distribuidos para el cálculo de la correspondencia y la construcción del grafo grueso.

- Paralelismo del tipo refinamiento. Es el tipo de paralelismo que se puede explotar durante la fase de refinamiento mediante el empleo de heurísticos paralelos distribuidos de optimización local para refinar el separador proyectado en cada uno de los niveles de la jerarquía de grafos.

La Figura 4.9 ejemplifica el marco general en el que se desarrolla el aprovechamiento de los dos últimos tipos paralelismo para el cálculo *paralelo* de separadores de vértices. En esta figura, el grafo más fino está distribuido entre $p = 4$ procesadores, y la construcción de la jerarquía de grafos se realiza en paralelo mediante algoritmos paralelos distribuidos para el cálculo de la correspondencia M y la construcción del grafo grueso. Los dos paquetes proporcionan las mismas opciones para distribuir el grafo grueso entre los procesadores. En la primera opción, que se puede observar en los primeros niveles de la Figura 4.9, el grafo grueso sigue distribuido entre p procesadores. Los vértices del grafo grueso construidos a partir de vértices locales del grafo fino, siguen siendo locales al procesador, lo que reduce el número medio de vértices por procesador y acelera las computaciones de niveles subsecuentes. Cuando en un determinado nivel de la jerarquía, el número medio de vértices locales por cada procesador es lo "suficientemente pequeño" como para no poder amortizar el sobrecoste de las comunicaciones entre p procesadores, entra en juego la segunda opción, mostrada en los últimos niveles de la Figura 4.9. El grafo grueso se *redistribuye* entre una primera mitad de $p/2$ procesadores y se *replica* en la otra mitad $p/2$ procesadores, lo que permite reducir el sobrecoste asociado a las comunicaciones en niveles subsecuentes. El proceso de engrosamiento, redistribución y réplica continúa hasta

que finalmente cada procesador recibe una copia del grafo grueso. A partir de ese momento, p procesadores calculan, de forma local e independiente, un separador del grafo grueso utilizando heurísticos secuenciales multinivel (ver parte inferior de la Figura 4.9). Esta solución, denominada optimización multisecuencial, no consigue acelerar el proceso de cálculo del separador del grafo grueso. Sin embargo, la experiencia adquirida [48] con los heurísticos multinivel indica que la calidad del separador se puede mejorar ejecutando múltiples instancias del algoritmo y seleccionando el mejor separador de entre todos los calculados, justificando la creación de varias copias del grafo grueso en cada uno de los procesadores. Cabe mencionar que la configuración por defecto de ParMETIS y PT-SCOTCH con respecto al proceso de redistribución y réplica difiere. En PT-SCOTCH, la redistribución y réplica comienza cuando el número medio de vértices del grafo grueso por cada procesador es inferior a 100. En ParMETIS, la redistribución y réplica está desactivada, y la fase de engrosamiento termina cuando el grafo más grueso tiene $O(p)$ nodos [104]. La parte paralela de la fase de refinamiento comienza cuando se alcanza el primer grafo distribuido de la jerarquía. Los procesadores entre los cuales está distribuido este grafo deciden cuál de los separadores calculados localmente tiene mejor calidad, proyectando sólo dicho separador. A continuación, este separador se refina en paralelo mediante empleo de heurísticos distribuidos de optimización local. La fase continúa en paralelo hasta llegar al grafo más fino.

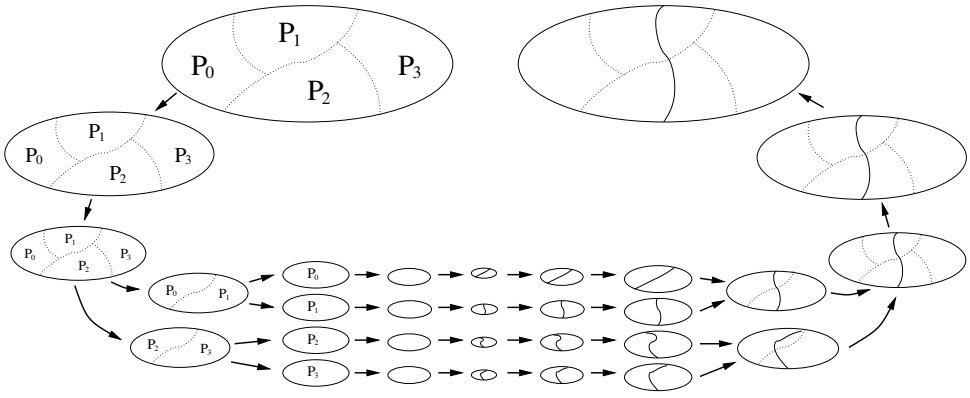


Figura 4.9: Diagrama de la computación paralela multinivel del separador de un grafo distribuido entre $p = 4$ procesadores, a través de engrosamiento paralelo con redistribución y réplica, computación multisecuencial del separador del grafo más grueso, y refinamiento paralelo del mejor separador calculado.

Paralelismo en la fase de engrosamiento

La parte más conflictiva de la fase de engrosamiento reside en el cálculo de M, y más concretamente, *en la frontera de los procesadores*. La frontera de un procesador está formada por aquellas aristas del grafo que conectan nodos pertenecientes a su partición (vértices locales) y nodos pertenecientes a particiones de otros procesadores (vértices remotos). Si para calcular M, cada procesador trabaja local e independientemente con su partición y frontera, entonces puede ocurrir que procesadores distintos seleccionen aristas que comparten un vértice, violando la propiedad de correspondencia de M. Por ejemplo, si los vértices v y u pertenecen a dos procesadores distintos, P_1 y P_2, entonces P_1 puede seleccionar la arista de su frontera que conecta v con u, mientras que P_2 puede seleccionar una arista que conecta u con un vértice diferente w. Además, otro procesador P_3 también puede seleccionar la arista que conecta un vértice z de su partición con u. En la literatura se han propuesto diversas estrategias para resolver este tipo de conflictos [48, 104]. ParMETIS y PT-SCOTCH adoptan, en su configuración por defecto, dos estrategias distintas:

- Correspondencia local (*local matching*). ParMETIS adopta la solución más simple y económica: *las aristas de las fronteras no pueden formar parte de* M. Por tanto, no es necesario que los procesadores se comuniquen entre sí para decidir qué aristas de las fronteras forman parte de M. Sin embargo, este enfoque no es robusto. La forma en la que está distribuido el grafo entre los procesadores determina el número de aristas que forman parte de la frontera de los procesadores. Si el grafo se distribuye de tal forma que hay un gran número aristas que forman parte de la frontera, entonces el enfoque sólo puede incorporar un pequeño número de aristas en M, afectando de forma significativa a su calidad. Por ejemplo, en dos de los grafos de prueba escogidos para el estudio experimental de [104] (en concreto, AUTO y MDUAL2), las fronteras de los procesadores contienen más del 90 % de las aristas de todo el grafo cuando se utiliza una distribución aleatoria o en bloques consecutivos de n/p vértices. Para aumentar la robustez de este enfoque, ParMETIS *redistribuye el grafo inicial explícitamente conforme a un particionado K-way* (*K-way partitioning*). El objetivo de este tipo de particionado es minimizar el número aristas que pertenecen a la frontera de los procesadores y, por tanto, el enfoque de correspondencias locales calcula correspondencias de muy buena calidad cuando se combina con particionados *K-way* [104]. Para calcular estos particionados, la rutina de disección anidada utiliza las rutinas *paralelas* multinivel de particionado *K-way* disponibles en ParMETIS [104].

- Correspondencia global (*global matching*). PT-SCOTCH emplea un heurístico de correspondencia global, es decir, un heurístico paralelo en el que las aristas de las fronteras pueden formar parte de M. En este caso, los procesadores se tienen que poner de acuerdo para decidir qué aristas de las fronteras forman parte de M. A continuación, esbozamos brevemente el principio de funcionamiento de este heurístico. Para simplificar la siguiente explicación, diremos que un vértice forma parte de M si alguna de sus aristas forma parte de M. Cada procesador mantiene una cola local que contiene aquellos vértices locales que todavía no forman parte de M, y repite los siguientes pasos. Desencolar el vértice situado en la cabeza de la cola y buscar, entre su lista de aristas, aquellas candidatas para formar parte de M. Una arista es candidata si conecta el vértice que nos ocupa con un vértice que no forme parte de M. Si no hay ninguna arista candidata, entonces el vértice forma parte directamente del grafo grueso (ya que ninguna de sus aristas puede formar parte de M). En caso contrario, el algoritmo selecciona una arista entre las candidatas (el criterio particular no es relevante para la explicación). Si esta arista conecta el vértice que nos ocupa con otro vértice local, entonces pasa a formar parte de M directamente. En caso contrario, el vértice local y el vértice remoto se bloquean temporalmente, es decir, ni la arista que los conecta puede formar parte de M (temporalmente), ni se pueden corresponder con otros vértices locales o remotos (temporalmente). Cuando ya no haya más nodos en la cola, entonces comienza la fase en la que se producen las comunicaciones. Cada procesador envía a cada uno de sus vecinos una lista con aquellos pares de vértices que han sido bloqueados localmente. Posteriormente, recibe las listas de sus vecinos, y las compara con sus listas locales. En concreto, por cada vecino, compara la lista local y la remota, y, por cada par de nodos de la lista local, busca el mismo par en la remota. Si lo encuentra, entonces la arista pasa a formar parte de M. En caso contrario, el vértice local y el remoto se desbloquean, y el vértice local se vuelve a introducir en la cola. Este proceso global se repite varias veces (iteraciones) hasta que finalmente el tamaño de las colas locales es lo "suficientemente pequeño".

Paralelismo en la fase de refinamiento

La diferencia más significativa entre ParMETIS y PT-SCOTCH reside en la fase de refinamiento, y más concretamente, en la estrategia utilizada para refinar los separadores en cada uno de los niveles de la jerarquía de grafos. La paralelización de heurísticos de optimización local,

como KL [106] o FM [69], resulta compleja debido a su naturaleza inherentemente secuencial. Un algoritmo paralelo que siga estrictamente el principio de funcionamiento de estos heurísticos requeriría comunicaciones cada vez que un procesador mueve un vértice local del separador que tiene vecinos en las particiones de otros procesadores. ParMETIS emplea un heurístico de optimización local *distribuido* [104], basado en FM, para refinar en paralelo el separador proyectado. Sin embargo, con el propósito de relajar las fuertes restricciones secuenciales de FM y reducir el sobrecoste asociado a las comunicaciones, *no permite movimientos que igualen o empeoren la calidad del separador si el vértice del separador involucrado en el movimiento tiene vecinos remotos.* Aunque este diseño del método paralelo permite acelerar el proceso de refinamiento del separador, inhabilita en cierto grado la capacidad del heurístico FM de escapar de mínimos locales, y puede afectar de forma significativa a la calidad del separador refinado cuando el número de procesadores aumenta (es decir, cuando aumenta la probabilidad de que un vértice local del separador tenga vecinos remotos). Esta es la causa por la que típicamente se observa una disminución de la calidad de la disección anidada paralela de ParMETIS cuando se utiliza como estrategia de reordenación para la reducción de llenado de la factorización dispersa directa de Cholesky [48].

La fase de refinamiento en PT-SCOTCH, al contrario que la de ParMETIS, no emplea heurísticos de optimización local distribuidos para refinar en paralelo el separador proyectado. En su lugar, ha sido diseñada para habilitar la aplicación de heurísticos *secuenciales* sobre grafos de gran dimensión (es decir, sobre grafos que no caben potencialmente en la memoria de un sólo procesador). Para ello, se basa en la siguiente observación sobre el comportamiento de los heurísticos de optimización local: cuando estos heurísticos se aplican sobre un separador inicial de calidad razonable (como el separador proyectado), tienden a perturbar sólo ligeramente su posición, refinándolo únicamente con vértices del grafo fino situados a "corta distancia" del separador inicial. La idea, por tanto, es aplicar algoritmos secuenciales de optimización local sobre un subgrafo del grafo fino que sólo contenga aquellos vértices "cercanos" al separador proyectado. Este subgrafo, *de tamaño significativamente más pequeño que el grafo fino*, se denomina grafo banda (*band graph*). Según [47], la aplicación de FM sobre grafos banda que contienen aquellos vértices del grafo fino situados *a distancia tres o inferior* del separador proyectado, no sólo permite mantener la calidad del separador del grafo más fino (es decir, el resultado del algoritmo), sino también mejorarla en muchos casos.

La construcción y uso de grafos banda distribuidos se ilustra en la Figura 4.10. En esta figura, el grafo fino está distribuido entre $p = 4$ procesadores, y el separador de vértices del grafo inmediatamente más grueso de la jerarquía ya ha sido proyectado sobre el grafo fino (ver grafo y separador en la parte izquierda de la Figura 4.10). La construcción del grafo banda distribuido (ver grafo situado en la parte central de la Figura 4.10) se inicia con el cálculo de las distancias de los vértices del grafo fino a los vértices del separador, utilizando para ello el algoritmo distribuido presentando en [47]. Posteriormente, para crear su partición local del grafo banda, cada procesador selecciona los vértices locales del grafo fino cuya distancia sea igual o inferior a un umbral preestablecido por el usuario, y añade, por cada uno de los subgrafos independientes separados, un vértice especial, denominado "vértice ancla". El vértice ancla es un artificio que permite aplicar el heurístico FM sobre el grafo banda sin tener que modificarlo para adaptarse a la condición especial de este grafo (especial porque realmente hay vértices más allá de la banda). El vértice ancla está conectado a los vértices locales de la última capa del grafo banda que también forma parte del grafo fino (es decir, aquellos vértices locales del grafo fino cuya distancia sea igual al umbral), y su peso es igual a la suma de los pesos de los vértices locales del subgrafo independiente correspondiente a los que substituye. Por tanto, debido a su gran peso, el heurístico FM no lo escoge como candidato para formar parte del separador. Por ejemplo, en la parte central de la Figura 4.10, el grafo banda está distribuido entre $p = 4$ procesadores. La partición del grafo banda correspondiente a P_0 y P_3 está vacía, ya que ninguno de estos dos procesadores tienen vértices locales del grafo fino situados a distancia igual o inferior

del separador. Por otra parte, los procesadores P_1 y P_2 han creado su partición local del grafo banda con aquellos vértices locales del grafo fino situados a distancia inferior o igual al umbral. Además, en la figura se puede observar que la partición de los procesadores P_1 y P_2 contiene un vértice ancla por cada subgrafo independiente, vértices que están conectados a la última capa del grafo banda que también forma parte del grafo fino, representada en la figura a través de círculos de color negro. Una vez que el separador del grafo banda se ha refinado utilizando algún heurístico de optimización local, éste se proyecta sobre el grafo fino (ver parte derecha de la Figura 4.10).

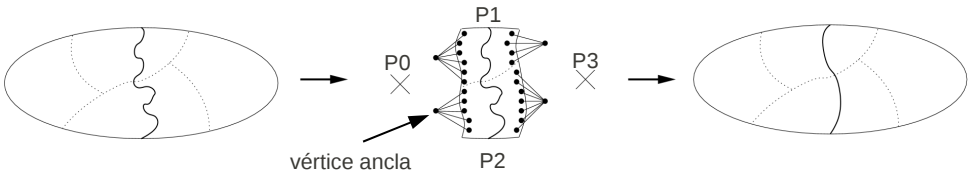


Figura 4.10: Construcción del grafo banda a partir del grafo fino y su separador. Una vez que el separador del grafo banda se ha refinado utilizando algún heurístico de optimización local, éste se proyecta sobre el grafo fino.

El refinamiento del separador del grafo banda procede como se ilustra en la Figura 4.11. Cada uno de los procesadores entre los cuales este grafo está distribuido, reúne (*gather*) las particiones de otros procesadores para crear una copia local del grafo banda completo. Posteriormente, cada procesador aplica el heurístico FM secuencial sobre su copia local para refinar el separador del grafo banda. La perturbación del estado inicial del algoritmo FM secuencial en cada procesador permite explorar en paralelo espacios de soluciones ligeramente diferentes, mejorando la calidad del separador refinado. Finalmente, el mejor separador calculado localmente se proyecta sobre el grafo fino distribuido. Esta solución, conocida como refinamiento multisecuencial [48], permite refinar separadores de grafos de gran escala sin pérdida significativa de calidad al aumentar el número de procesadores. Sin embargo, no es escalable en términos de eficiencia paralela, dado que no acelera el proceso de refinamiento al aumentar el número de procesadores.

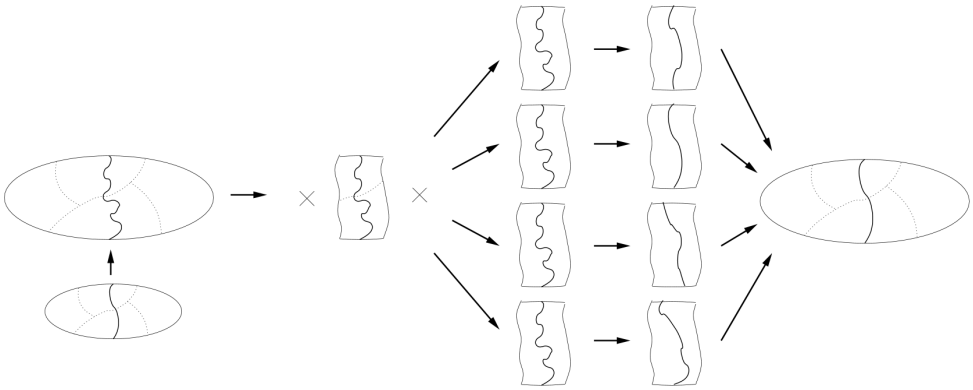


Figura 4.11: Refinamiento multisecuencial del separador del grafo fino proyectado a partir del grafo inmediatamente más grueso de la jerarquía.

4.5.2. Evaluación experimental

En este apartado evaluamos la aceleración que se puede lograr en el proceso de resolución de (4.1) utilizando las dos aproximaciones paralelas revisadas en el Apartado 4.5.1. Asimismo, estudiamos el consumo de memoria de la disección anidada paralela que incorporan estas solu-

ciones, y el impacto que tiene el particionado calculado por éstas sobre la eficiencia del cálculo paralelo del precondicionador LDL^T multinivel y la resolución iterativa del sistema.

Cambios en la configuración de los algoritmos y entorno de ejecución paralelos

A continuación, enumeramos los cambios realizados en la configuración de los algoritmos y entorno de ejecución paralelos (consultar la Sección 4.2) para obtener los resultados recogidos en este apartado:

- La implementación del Algoritmo 2.20 corre ahora a cargo de las bibliotecas paralelas de particionado de grafos PT-SCOTCH, versión 5.0.6RC16, o ParMETIS, versión 1.1.1.
- `ND-HAMD-B` es la estrategia escogida (en lugar de `ND-HAMD-A`) para la fase inicial de particionado, dado que ésta consiguió reducir en mayor medida el tiempo de ejecución de todo el proceso de resolución (consultar la Sección 4.3).
- Los códigos paralelos de las bibliotecas PT-SCOTCH y ParMETIS se han enlazado con una versión de la biblioteca MPI optimizada para la arquitectura con memoria compartida del multiprocesador SGI Altix 350. Esta biblioteca forma parte del paquete SGI MPT (*Message Passing Tools*), versión 1.12, que el fabricante ha desarrollado para implementar eficientemente el modelo de programación paralelo basado en el paso de mensajes en este multiprocesador.
- El entorno de ejecución de MPI se ha configurado de tal forma que ligamos un proceso MPI por cada procesador físico del multiprocesador, y evitamos que éstos migren durante la ejecución de los códigos paralelos de PT-SCOTCH y ParMETIS.

Descripción del experimento y presentación de los resultados

El primer experimento de este apartado compara el desempeño de PT-SCOTCH y ParMETIS para el cálculo paralelo de separadores de vértices en la etapa inicial de particionado, y evalúa el impacto de los particionados calculados por estos paquetes sobre la eficiencia de la etapas numéricas de nuestro resolutor. La altura (número de niveles) y el número de hojas del árbol de tareas se fijaron durante la etapa inicial de particionado como una función de p, *utilizando la estrategia "$f = c$" en todos los casos*. Por lo tanto, los árboles binarios obtenidos son de altura $log_2(p) + 1$ y completos, es decir, con p hojas. El experimento no explora valores de f en el intervalo $[p, 3p]$ (como en el experimento de la Sección 4.3) debido a que el número de procesadores involucrados en la disección anidada paralela de ParMETIS sólo puede ser potencia de dos y, además, el número de subgrafos independientes calculados por el proceso debe coincidir con el número de procesadores. PT-SCOTCH no tiene esta limitación, y cualquier número arbitrario de procesadores puede calcular cualquier número arbitrario de particiones. Sin embargo, para comparar ambas soluciones bajo las mismas condiciones, el experimento no explora esta posibilidad de PT-SCOTCH. Las mediciones se realizaron conforme a la siguiente estrategia. Para cada combinación de matriz, número de procesadores y biblioteca, ejecutamos diez veces la terna formada por las tres etapas del resolutor paralelo. En cada una de estas ejecuciones, registramos el tiempo de ejecución total (es decir, la suma de los tiempos de ejecución de las tres etapas), constituyendo una muestra con 10 resultados. Finalmente, se determina la ejecución correspondiente a la mediana de la muestra, considerando únicamente las mediciones tomadas en dicha ejecución

En las Tablas 4.8, 4.9 y 4.10 comparamos el rendimiento del resolutor secuencial incluido en la biblioteca ILUPACK (resultados para $p = 1$ en las tablas) con el de nuestro resolutor paralelo utilizando PT-SCOTCH o ParMETIS para implementar la etapa inicial de particionado, y $p = 2$, 4, 8 o 16 procesadores para calcular en paralelo las tres etapas involucradas en el proceso de resolución de los 12 sistemas de la Tabla 4.1. En concreto, la Tabla 4.8 presenta los resultados

para los sistemas de menor tamaño (VC, C, M y F), la Tabla 4.9, para los de tamaño intermedio (VC2, M1, VF y F1) y, por último, la Tabla 4.10, para los de mayor tamaño (C2, VC3, M2 y VF1). Los resultados para cada matriz se separan en dos grupos (de cinco filas cada uno) en función de la biblioteca paralela utilizada para realizar los cálculos de la etapa inicial de particionado. La columna ubicada debajo de la etiqueta "Total" muestra el tiempo de ejecución total del resolutor, y las tablas muestran resultados separados para cada una de las tres etapas.

Cód.	p	Total	Disecc. Anid.		Precond. LDL^T Multiniv.			Método PCG			
		T_p (seg.)	Opc. ND-HAMD-B	T_p (seg.)	nnz_M (mill.)	T_p (seg.)	S_p (T_1/T_p)	#Iter.	Err.	T_p (seg.)	S_p (T_1/T_p)
VC	1	0,02	B-PS	0,00	0,0	0,02	1,0	11	2,3e-15	0,01	1,0
	2	0,02	B-PS	0,01	0,0	0,01	1,8	11	1,6e-14	0,01	1,3
	4	0,02	B-PS	0,01	0,0	0,01	2,6	12	2,0e-15	0,01	1,0
	8	0,03	B-PS	0,01	0,0	0,01	2,5	12	1,2e-14	0,01	0,6
	16	0,05	B-PS	0,02	0,0	0,01	1,8	12	2,3e-14	0,02	0,4
	1	0,02	B-PM	0,00	0,0	0,02	1,0	11	2,3e-15	0,01	1,0
	2	0,03	B-PM	0,01	0,0	0,01	1,8	12	6,0e-16	0,01	1,2
	4	0,03	B-PM	0,02	0,0	0,01	2,5	12	3,2e-15	0,01	1,0
	8	0,04	B-PM	0,02	0,0	0,01	2,5	12	9,6e-15	0,01	0,6
	16	0,06	B-PM	0,04	0,0	0,01	1,8	12	8,3e-14	0,02	0,4
C	1	0,18	B-PS	0,00	0,1	0,13	1,0	14	8,2e-13	0,05	1,0
	2	0,14	B-PS	0,04	0,1	0,07	1,8	14	9,6e-13	0,03	1,6
	4	0,13	B-PS	0,07	0,1	0,04	3,1	14	6,6e-13	0,02	2,3
	8	0,13	B-PS	0,08	0,1	0,03	4,3	15	8,4e-13	0,02	2,2
	16	0,17	B-PS	0,11	0,1	0,03	4,9	15	6,0e-13	0,03	1,7
	1	0,18	B-PM	0,00	0,1	0,13	1,0	14	8,2e-13	0,05	1,0
	2	0,14	B-PM	0,04	0,1	0,07	1,8	15	2,0e-13	0,03	1,6
	4	0,12	B-PM	0,05	0,1	0,04	3,0	15	3,0e-13	0,02	2,2
	8	0,11	B-PM	0,06	0,1	0,03	4,1	15	1,1e-12	0,02	2,2
	16	0,13	B-PM	0,07	0,1	0,03	4,4	15	8,3e-13	0,03	1,8
M	1	0,89	B-PS	0,00	0,4	0,60	1,0	17	8,5e-12	0,30	1,0
	2	0,57	B-PS	0,09	0,4	0,33	1,8	17	1,4e-11	0,15	2,0
	4	0,46	B-PS	0,21	0,4	0,17	3,6	18	5,4e-12	0,08	3,6
	8	0,39	B-PS	0,22	0,4	0,11	5,5	18	1,0e-11	0,06	4,7
	16	0,46	B-PS	0,32	0,4	0,09	6,8	18	1,4e-11	0,06	5,2
	1	0,89	B-PM	0,00	0,4	0,60	1,0	17	8,5e-12	0,30	1,0
	2	0,57	B-PM	0,13	0,4	0,31	1,9	17	1,6e-11	0,14	2,2
	4	0,38	B-PM	0,12	0,4	0,17	3,5	18	6,2e-12	0,09	3,5
	8	0,29	B-PM	0,11	0,4	0,11	5,2	18	7,7e-12	0,07	4,5
	16	0,32	B-PM	0,15	0,4	0,11	5,6	18	1,1e-11	0,06	4,7
F	1	4,39	B-PS	0,00	1,5	2,61	1,0	23	5,0e-11	1,77	1,0
	2	2,41	B-PS	0,28	1,5	1,32	2,0	22	1,2e-10	0,81	2,2
	4	1,74	B-PS	0,53	1,5	0,77	3,4	22	1,2e-10	0,44	4,0
	8	1,37	B-PS	0,66	1,5	0,46	5,7	23	1,0e-10	0,26	6,9
	16	1,17	B-PS	0,69	1,5	0,30	8,7	23	1,2e-10	0,18	10,1
	1	4,39	B-PM	0,00	1,5	2,61	1,0	23	5,0e-11	1,77	1,0
	2	2,60	B-PM	0,43	1,5	1,32	2,0	23	8,4e-11	0,85	2,1
	4	1,40	B-PM	0,32	1,5	0,69	3,8	22	1,3e-10	0,38	4,6
	8	0,91	B-PM	0,26	1,5	0,41	6,3	23	8,9e-11	0,24	7,5
	16	0,75	B-PM	0,25	1,5	0,31	8,3	23	6,2e-11	0,19	9,4

Tabla 4.8: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Para la etapa inicial de particionado se utilizó la estrategia ND-HAMD-B implementada mediante PT-SCOTCH (B-PS) o PARMETIS (B-PM). Los resultados de esta tabla se refieren a los cuatro sistemas de menor tamaño, es decir, VC, C, M y F en la Tabla 4.1.

Cód.	p	Total T_p (seg.)	Disecc. Anid. Opc. ND-HAMD-B	Disecc. Anid. T_p (seg.)	Precond. LDL^T Multiniv. nnz_M (mill.)	Precond. LDL^T Multiniv. T_p (seg.)	Precond. LDL^T Multiniv. S_p (T_1/T_p)	Método PCG #Iter.	Método PCG Err.	Método PCG T_p (seg.)	Método PCG S_p (T_1/T_p)
VC2	1	16,07	B	0,00	4,5	9,00	1,0	28	4,3e-10	7,07	1,0
	2	8,88	B-PS	1,30	4,4	4,19	2,1	29	3,7e-10	3,39	2,1
	4	4,89	B-PS	1,04	4,5	2,13	4,2	29	4,1e-10	1,72	4,1
	8	3,38	B-PS	1,22	4,5	1,23	7,3	29	4,3e-10	0,93	7,6
	16	2,69	B-PS	1,38	4,5	0,75	11,9	29	4,9e-10	0,56	12,7
	1	16,07	B-PM	0,00	4,5	9,00	1,0	28	4,3e-10	7,07	1,0
	2	8,98	B-PM	1,28	4,5	4,26	2,1	29	3,0e-10	3,45	2,0
	4	4,78	B-PM	0,97	4,5	2,18	4,1	28	6,7e-10	1,63	4,3
	8	2,73	B-PM	0,60	4,5	1,20	7,5	29	3,7e-10	0,93	7,6
	16	1,82	B-PM	0,48	4,5	0,78	11,6	29	4,3e-10	0,56	12,6
M1	1	17,43	B	0,00	4,7	9,77	1,0	28	1,3e-10	7,66	1,0
	2	9,83	B-PS	0,82	4,7	5,06	1,9	28	2,3e-10	3,95	1,9
	4	5,10	B-PS	1,00	4,8	2,32	4,2	28	3,9e-10	1,79	4,3
	8	3,63	B-PS	1,25	4,8	1,38	7,1	28	4,5e-10	1,01	7,6
	16	2,87	B-PS	1,51	4,8	0,80	12,2	28	3,2e-10	0,56	13,8
	1	17,43	B	0,00	4,7	9,77	1,0	28	1,3e-10	7,66	1,0
	2	10,10	B-PM	1,47	4,7	4,90	2,0	28	1,6e-10	3,73	2,1
	4	5,19	B-PM	1,03	4,8	2,36	4,1	28	3,6e-10	1,79	4,3
	8	3,04	B-PM	0,67	4,8	1,35	7,2	28	3,8e-10	1,01	7,6
	16	2,07	B-PM	0,53	4,9	0,90	10,9	28	2,1e-10	0,65	11,8
VF	1	47,25	B	0,00	10,8	24,82	1,0	34	5,8e-10	22,43	1,0
	2	27,51	B-PS	3,27	10,9	12,63	2,0	35	4,2e-10	11,61	1,9
	4	13,98	B-PS	2,68	11,0	5,96	4,2	34	9,5e-10	5,34	4,2
	8	8,85	B-PS	2,84	11,0	3,16	7,9	35	4,7e-10	2,85	7,9
	16	6,53	B-PS	2,83	11,1	1,95	12,8	34	1,0e-09	1,75	12,8
	1	47,25	B	0,00	10,8	24,82	1,0	34	5,8e-10	22,43	1,0
	2	27,43	B-PM	3,53	10,9	12,69	2,0	34	7,1e-10	11,21	2,0
	4	14,29	B-PM	2,36	11,0	6,32	3,9	34	1,1e-09	5,61	4,0
	8	7,90	B-PM	1,54	11,0	3,44	7,2	34	7,0e-10	2,92	7,7
	16	4,79	B-PM	1,08	11,1	1,97	12,6	34	8,5e-10	1,74	12,9
F1	1	69,65	B	0,00	15,2	36,03	1,0	37	9,9e-10	33,62	1,0
	2	39,49	B-PS	4,11	14,9	17,88	2,0	38	1,0e-09	17,50	1,9
	4	21,26	B-PS	3,85	14,9	8,72	4,1	38	9,0e-10	8,68	3,9
	8	12,79	B-PS	3,87	14,9	4,52	8,0	38	1,9e-09	4,39	7,7
	16	9,31	B-PS	4,02	15,0	2,65	13,6	38	2,1e-09	2,64	12,7
	1	69,65	B	0,00	15,2	36,03	1,0	37	9,9e-10	33,62	1,0
	2	39,25	B-PM	5,00	14,9	17,34	2,1	38	8,5e-10	16,91	2,0
	4	21,22	B-PM	3,30	14,9	9,03	4,0	38	1,5e-09	8,89	3,8
	8	11,04	B-PM	2,10	15,0	4,49	8,0	38	1,1e-09	4,45	7,6
	16	6,47	B-PM	1,43	15,1	2,58	14,0	38	1,9e-09	2,47	13,6

Tabla 4.9: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Para la etapa inicial de particionado se utilizó la estrategia `ND-HAMD-B` implementada mediante PT-SCOTCH (`B-PS`) o PARMETIS (`B-PM`). Los resultados de esta tabla se refieren a los cuatro sistemas de tamaño intermedio, es decir, VC2, M1, VF y F1 en la Tabla 4.1.

Evaluación del desempeño de ambos paquetes para el cálculo paralelo de la disección anidada

Las dos columnas etiquetadas como "Disecc. Anid." en las Tablas 4.8, 4.9 y 4.10, es decir, "Opc." y "T_p", muestran, respectivamente, el paquete paralelo usado para particionar el grafo de adyacencia (`B-PS`, para PT-SCOTCH, y `B-PM`, para ParMETIS), y el tiempo de ejecución paralelo en segundos empleado por el paquete correspondiente para dicho cálculo. La ejecución de esta etapa es paralela y, por tanto, p no sólo se utiliza para ajustar el grado de paralelismo disponible durante el cálculo del precondicionador LDL^T multinivel y la resolución iterativa

Cód.	p	Total T_p (seg.)	Disecc. Anid. Opc. ND-HAMD-B	Disecc. Anid. T_p (seg.)	Precond. LDL^T nnz_M (mill.)	Precond. LDL^T T_p (seg.)	Multiniv. S_p (T_1/T_p)	Método PCG #Iter.	Método PCG Err.	Método PCG T_p (seg.)	Método PCG S_p (T_1/T_p)
C2	1	73,62	B	0,00	15,9	35,77	1,0	40	1,9e-09	37,85	1,0
	2	42,93	B-PS	4,77	16,2	19,11	1,9	39	2,2e-09	19,05	2,0
	4	22,61	B-PS	4,82	16,0	8,67	4,1	40	2,2e-09	9,13	4,1
	8	13,71	B-PS	3,77	16,1	4,98	7,2	40	2,4e-09	4,95	7,6
	16	9,76	B-PS	3,97	16,2	2,79	12,8	40	2,5e-09	3,00	12,6
	1	73,62	B	0,00	15,9	35,77	1,0	40	1,9e-09	37,85	1,0
	2	42,75	B-PM	5,12	16,1	18,66	1,9	39	2,4e-09	18,97	2,0
	4	22,41	B-PM	3,53	16,2	9,23	3,9	40	2,0e-09	9,65	3,9
	8	11,79	B-PM	2,11	16,1	4,85	7,4	40	1,6e-09	4,84	7,8
	16	6,87	B-PM	1,44	16,2	2,68	13,4	40	2,1e-09	2,76	13,7
VC3	1	247,05	B	0,00	45,2	114,30	1,0	50	4,8e-09	132,75	1,0
	2	134,35	B-PS	12,15	45,3	55,15	2,1	50	4,8e-09	67,05	2,0
	4	75,81	B-PS	14,67	45,3	26,93	4,2	50	4,7e-09	34,21	3,9
	8	46,33	B-PS	14,19	45,5	14,50	7,9	50	5,1e-09	17,65	7,5
	16	31,23	B-PS	12,89	45,2	7,93	14,4	52	3,6e-09	10,41	12,8
	1	247,05	B	0,00	45,2	114,30	1,0	50	4,8e-09	132,75	1,0
	2	140,53	B-PM	17,57	45,2	54,28	2,1	51	3,7e-09	68,68	1,9
	4	74,73	B-PM	9,88	45,2	28,41	4,0	51	4,3e-09	36,43	3,6
	8	39,98	B-PM	5,95	45,2	15,02	7,6	51	4,7e-09	19,01	7,0
	16	22,02	B-PM	3,77	45,2	7,89	14,5	52	3,4e-09	10,36	12,8
M2	1	262,03	B	0,00	47,7	122,33	1,0	50	2,1e-09	139,70	1,0
	2	144,69	B-PS	13,11	47,5	59,75	2,0	50	3,0e-09	71,82	1,9
	4	79,96	B-PS	14,37	47,2	28,87	4,2	50	3,4e-09	36,71	3,8
	8	47,76	B-PS	12,44	47,4	15,90	7,7	51	1,9e-09	19,42	7,2
	16	29,04	B-PS	10,27	47,3	8,15	15,0	51	2,3e-09	10,62	13,2
	1	262,03	B	0,00	47,7	122,33	1,0	50	2,1e-09	139,70	1,0
	2	152,09	B-PM	19,65	47,1	58,76	2,1	51	1,9e-09	73,68	1,9
	4	77,90	B-PM	11,10	47,2	28,82	4,2	51	2,8e-09	37,98	3,7
	8	41,90	B-PM	6,48	47,4	15,85	7,7	49	3,7e-09	19,57	7,1
	16	23,66	B-PM	4,30	47,5	8,75	14,0	51	2,0e-09	10,60	13,2
VF1	1	705,09	B	0,00	100,7	287,40	1,0	60	7,7e-09	417,68	1,0
	2	364,13	B-PS	35,15	99,7	142,58	2,0	61	6,4e-09	186,40	2,2
	4	197,50	B-PS	34,97	99,4	66,26	4,3	62	4,7e-09	96,27	4,3
	8	113,59	B-PS	29,34	98,8	33,81	8,5	62	5,5e-09	50,43	8,3
	16	71,06	B-PS	23,05	99,1	18,91	15,2	62	6,0e-09	29,10	14,4
	1	705,09	B	0,00	100,7	287,40	1,0	60	7,7e-09	417,68	1,0
	2	397,63	B-PM	59,13	100,2	147,48	1,9	61	5,8e-09	191,02	2,2
	4	208,17	B-PM	33,86	100,0	73,35	3,9	62	4,7e-09	100,97	4,1
	8	110,37	B-PM	19,55	99,3	36,43	7,9	62	5,7e-09	54,40	7,7
	16	60,69	B-PM	11,31	99,0	19,24	14,9	62	5,7e-09	30,14	13,9

Tabla 4.10: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución numérica paralela de (4.1). Para la etapa inicial de particionado se utilizó la estrategia `ND-HAMD-B` implementada mediante PT-SCOTCH (`B-PS`) o PARMETIS (`B-PM`). Los resultados de esta tabla se refieren a los cuatro sistemas de mayor tamaño, es decir, C2, VC3, M2 y VF1 en la Tabla 4.1.

del sistema, sino que además indica el número de procesadores involucrados en el cálculo de la disección anidada. Es necesario remarcar que al duplicar p se calcula un nivel adicional de recursividad de la disección anidada y, por tanto, el tiempo de ejecución de la etapa inicial de particionado puede aumentar con p. Este aumento se produce cuando el sobrecoste asociado al cálculo de un nivel adicional de la disección anidada no se puede amortizar con la aceleración que supone emplear el doble de procesadores para el cálculo de este nivel adicional y todos los anteriores. Los resultados de las tablas confirman que la paralelización propuesta en ParMETIS para la disección anidada multinivel es más eficiente que la de PT-SCOTCH. Excepto en el

caso de los tres sistemas de menor dimensión (es decir, VC, C y M en la Tabla 4.8), el tiempo de ejecución paralelo de ParMETIS siempre se redujo al aumentar p, mientras que en el caso de PT-SCOTCH, se redujo en menor grado, se mantuvo aproximadamente constante e incluso aumentó dependiendo del sistema y del número de procesadores. Por ejemplo, en el caso del sistema VC3 (Tabla 4.10), el tiempo de ejecución de ParMETIS fue de 17,57, 9,88, 5,95 y 3,77 segundos para $p = 2$, 4, 8 y 16 procesadores, respectivamente, mientras que en el caso de PT-SCOTCH fue de 12,15, 14,67, 14,19, y 12,89 segundos para el mismo número de procesadores; es decir, aumentó de 2 a 4 procesadores, se mantuvo aproximadamente constante de 4 a 8, y se redujo en menor medida de 8 a 16 (una reducción del 9 % frente a una del 36 % en el caso de ParMETIS). La mayor eficiencia paralela de ParMETIS frente a PT-SCOTCH se debe principalmente a las diferentes técnicas y algoritmos utilizados por ambos paquetes para aprovechar paralelismo durante las fases de engrosamiento y refinamiento. Según el Apartado 4.5.1, ParMETIS realiza dos compromisos sobre la calidad de los separadores con el propósito de aumentar su eficiencia paralela (como veremos más adelante en este apartado, estos compromisos afectan negativamente a la calidad del particionado cuando éste se utiliza para calcular una reordenación para reducción de llenado de la factorización directa de Cholesky dispersa). En primer lugar, en la fase de engrosamiento utiliza un heurístico de correspondencia local para calcular M, reduciendo el sobrecoste de las comunicaciones frente al de correspondencia global que incorpora PT-SCOTCH. En segundo lugar, en la fase de refinamiento utiliza un heurístico distribuido de optimización local que no permite cierto tipo de movimientos con el propósito de reducir el sobrecoste de las comunicaciones durante el refinamiento paralelo del separador. Sin embargo, PT-SCOTCH utiliza refinamiento multisecuencial y, por tanto, el coste de la fase de refinamiento no se reduce con p, lo que supone un cuello de botella secuencial para la eficiencia paralela de la disección anidada paralela. Teniendo en cuenta que el coste relativo de la fase de refinamiento con respecto a todo el proceso de disección oscila aproximadamente entre el 30 % y el 45 % para los sistemas de prueba considerados, no es sorprendente la pobre aceleración obtenida por PT-SCOTCH al aumentar p.

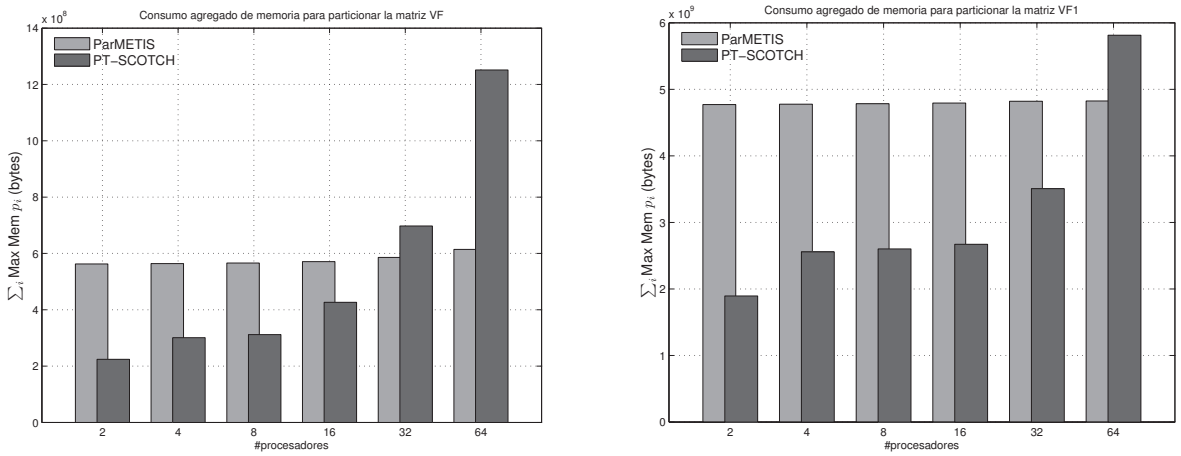
Los resultados de las tablas revelan que el tiempo de ejecución de PT-SCOTCH fue (generalmente) inferior al de ParMETIS con $p = 2$ procesadores. Esto se debe al sobrecoste que en ParMETIS supone calcular y redistribuir de antemano el grafo de adyacencia conforme a un particionado *K-way* (consultar el Apartado 4.5.1). Por ejemplo, en el caso de las matrices VC3 y VF1, y $p = 2$ procesadores, hemos comprobado que el tiempo de ejecución empleado para calcular el particionado *K-way* fue de 8,9 y 35,3 segundos, respectivamente, es decir, ligeramente superior al empleado por la disección anidada para calcular el separador raíz (Tabla 4.10). Sin embargo, al aumentar p, el sobrecoste del particionado *K-way* se puede amortizar con el mayor grado paralelismo explotado durante la fase de refinamiento, superando a PT-SCOTCH, especialmente con $p = 8$ y 16 procesadores. Por ejemplo, en el caso de la matriz F1 (Tabla 4.10) y con $p = 8$ y 16 procesadores, el tiempo de ejecución de PT-SCOTCH fue de 3,87 y 4,02 segundos, mientras que el de ParMETIS fue de 2,10 y 1,43 segundos; se puede observar el mismo comportamiento del rendimiento relativo de ambos paquetes para el resto de sistemas. Esperamos que las diferencias sean todavía más significativas al aumentar el número de procesadores por encima de 16.

Evaluación de la memoria consumida para el cálculo paralelo de la disección anidada

El siguiente experimento compara el consumo de memoria de ambos paquetes para calcular los primeros $log_2(p) + 1$ niveles de recursividad de la disección anidada (es decir, el número de niveles de recursividad que `ND-HAMD-B` calcula para p procesadores). Para ello, hemos instrumentado los módulos de memoria de PT-SCOTCH y ParMETIS de tal forma que cada proceso MPI registra de forma independiente al resto, la memoria consumida localmente en cada instante de tiempo. En el proceso de construcción de un separador, la memoria consumida aumenta durante la fase de engrosamiento cada vez que se construye un nuevo grafo grueso de la jerarquía. En el

caso de PT-SCOTCH, la memoria consumida también aumenta durante la fase de refinamiento, debido a la creación de las copias del grafo banda en cada uno de los procesadores involucrados en el refinamiento multisecuencial; la memoria necesaria para almacenar estas copias se libera tan pronto como se ha proyectado el mejor separador refinado localmente sobre el grafo fino distribuido (ver Figura 4.11). Una vez calculado el separador del grafo más fino, y creadas las estructuras de datos para almacenar los subgrafos independientes separados, la memoria consumida por ambos paquetes disminuye al liberar la memoria consumida por la jerarquía de grafos empleada para calcular el mencionado separador. Cuando termina la ejecución paralela de la etapa inicial de particionado, cada proceso determina la máxima memoria consumida localmente, y los procesos se sincronizan para calcular la suma de todos los máximos locales, constituyendo una sóla métrica, *la memoria agregada consumida por el método paralelo de particionado*.

En la Figura 4.12 mostramos la memoria agregada consumida por PT-SCOTCH (barras en gris) y ParMETIS (barras en negro) como una función del número de procesadores. La Figura 4.12 (a) se centra en el sistema F1, y la Figura 4.12 (b) en el sistema VF1. Como se puede observar en estas figuras, ParMETIS presenta escalabilidad de memoria cuasi-lineal, dado que la memoria agregada sólo crece ligeramente con p. Sin embargo, aunque PT-SCOTCH consume inicialmente menos memoria, ésta crece de forma significativa con p, superando la memoria consumida por ParMETIS para los dos sistemas de las Figuras 4.12 (a) y (b). Al analizar minuciosamente las trazas generadas por nuestra instrumentación, hemos comprobado que el consumo máximo de memoria de PT-SCOTCH se produce durante el segundo nivel de recursividad de la disección anidada, y más concretamente, inmediatamente después de reservar memoria para las copias del grafo banda necesarias para refinar el separador del grafo más fino de la jerarquía. Por lo tanto, no es sorprendente el crecimiento de la memoria de PT-SCOTCH con p, dado que el número de copias del grafo banda aumenta linealmente con p. Lo que resulta más sorprendente (por negativo) es que la memoria necesaria para almacenar estas copias sea tan alta en relación a la memoria consumida para almacenar toda la jerarquía de grafos (esta última memoria equivale aproximadamente a la memoria consumida por el método con $p = 2$ procesadores). El comportamiento observado para estos dos sistemas es representativo del observado para el resto de los sistemas de la Tabla 4.1.



(a) (b)

Figura 4.12: Memoria agregada consumida (bytes) por PT-SCOTCH y ParMETIS para particionar la matriz (a) F1 y (b) VF1 con $p = 2$, 4, 8, 16, 32 y 64 procesadores.

Efecto de los particionados sobre el rendimiento de las fases numéricas de nuestro resolutor

Las columnas etiquetadas como "Precond. LDL^T Multiniv." y "Método PCG" en las Tablas 4.8, 4.9 y 4.10 muestran, respectivamente, el rendimiento del cálculo paralelo del precondicio-

nador LDL^T multinivel y de la resolución iterativa del sistema, y su significado coincide con el de las columnas de las Tablas 4.2, 4.3 y 4.4. Como se puede observar en las tablas, el número de elementos no nulos del precondicionador, el número de iteraciones requeridas por el método iterativo, así como la precisión de la solución calculada por éste, presentan un grado muy bajo de sensibilidad al número de procesadores y a la biblioteca utilizada para calcular el particionado del grafo de adyacencia. Por ejemplo, en el caso del sistema C2, el número de elementos no nulos del precondicionador fue, con PT-SCOTCH y $p = 2$, 4, 8 y 16 procesadores, de 16,2, 16,0, 16,1 y 16,2 millones, respectivamente, mientras que con ParMETIS fue de 16,1, 16,2, 16,1 y 16,2 millones. Aunque el tiempo de ejecución paralelo y la aceleración paralela del cálculo del precondicionador y de la resolución iterativa presentan un mayor grado de sensibilidad al paquete empleado para calcular el particionado, las diferencias son moderadas en todos los casos, y nuestro enfoque de paralelización obtiene aceleraciones notables para las etapas numéricas con los particionados calculados por cualquiera de los dos paquetes. Además, el paquete que obtiene aquel particionado que consigue acelerar en mayor medida las etapas numéricas numéricas del resolutor, depende de cada matriz y del número de procesadores. Por ejemplo, en el caso del sistema C2, y $p = 16$ procesadores, la aceleración obtenida para el cálculo del precondicionador con el particionado de PT-SCOTCH, fue de 12,8, y de 13,4 con el de ParMETIS. Sin embargo, para el sistema M2, estas aceleraciones fueron de 15,0 y 14,0 con los particionados calculados por PT-SCOTCH y ParMETIS, respectivamente.

La resultados de las Tablas 4.8, 4.9 y 4.10 revelan que *el rendimiento de nuestro método numérico paralelo no se degrada con p, independientemente del paquete utilizado para el cálculo del particionado inicial* (en relación al rendimiento obtenido en las Tablas 4.2, 4.3 y 4.4 con el paquete secuencial SCOTCH). Esto de debe a que, con el valor típico de las tolerancias de descarte, como por ejemplo, $\tau = 10^{-2}$ o 10^{-3}, la factorización LDL^T con pivotamiento basado en la inversa descarta muchos elementos de los factores incompletos, especialmente para sistemas ligados a EDPs tridimensionales. Sin embargo, en el caso de los métodos directos sí que se observa esta degradación, y con mayor severidad en el caso de ParMETIS debido a las restricciones que impone este paquete sobre el heurístico distribuido de optimización local en la fase de refinamiento del separador [48]. Para cuantificar esta degradación en el caso de nuestro problema de aplicación, hemos determinado (mediante una factorización simbólica [56]) el número de operaciones en coma flotante requeridas para completar una factorización directa de Cholesky de la matriz $\Pi^T A \Pi$, donde A es la matriz del sistema original, y Π es una matriz de permutación para reducción de llenado calculada por la disección anidada multinivel paralela de ParMETIS o PT-SCOTCH, con $p = 2$, 4, 8 y 16 procesadores. En la Figura 4.13 ilustramos, para los sistemas de tamaño intermedio y de mayor tamaño (VC2, M1, VF, F1, C2, VC3, M2 y VF1 de izquierda a derecha y de arriba a abajo), el ratio entre el número de operaciones requeridas para factorizar la matriz reordenada mediante PT-SCOTCH (barras de color gris) y ParMETIS (barras de color negro) con $p = 2$, 4, 8 y 16 procesadores, y el número de operaciones requeridas para factorizar la matriz reordenada con la disección anidada multinivel *secuencial* de SCOTCH. Como se puede observar en la figura, el valor del ratio siempre aumenta con p para ParMETIS, en algunos casos de forma significativa, confirmando la degradación de la calidad de la reordenación calculada por ParMETIS con el número de procesadores. Por ejemplo, en el caso de la matriz M2, y $p = 16$ procesadores, la factorización de Cholesky dispersa requirió más del doble de operaciones con la reordenación calculada por ParMETIS. Sin embargo, en el caso de PT-SCOTCH, el valor del ratio crece de forma más moderada con p, se mantiene, e incluso decrece con p dependiendo del sistema. Por ejemplo, en el caso de la matriz F1 crece de forma más moderada, en el caso de la matriz C2 se mantiene aproximadamente, y en el caso de M1 decrece con p.

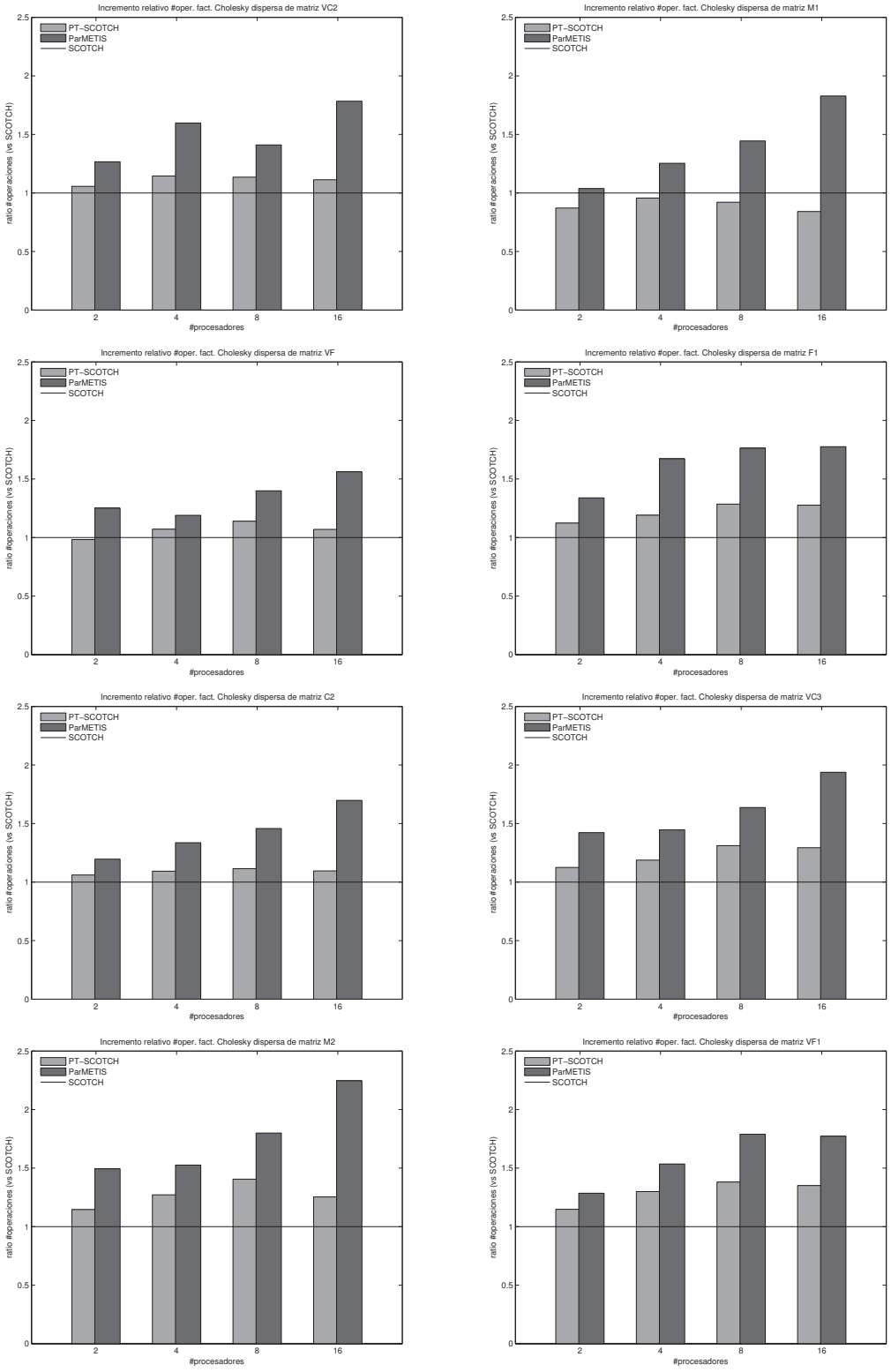


Figura 4.13: Ratio entre el número de operaciones en coma flotante necesarias para completar una factorización directa de Cholesky dispersa de la reordenación calculada por la disección anidada *paralela* de ParMETIS y PT-SCOTCH, y $p = 2, 4, 8$ o 16 procesadores, y el número de operaciones con la reordenación calculada por la disección anidada *secuencial* de SCOTCH.

4.6. Evaluación experimental con una miscelánea de sistemas

En esta sección evaluamos el enfoque de paralelización con una variedad de sistemas obtenidos de la colección de matrices dispersas de la Universidad de Florida [55]. El objetivo de la evaluación es demostrar experimentalmente que el enfoque de paralelización también es eficiente para sistemas de ecuaciones irregulares, provenientes de diversas áreas de aplicación.

Miscelánea de sistemas

La Tabla 4.11 presenta, para cada uno de los 14 sistemas de prueba escogidos para el estudio experimental, su código, nombre, área de la aplicación de origen (o descripción), número de filas/columnas, número de entradas no nulas y número medio de entradas no nulas por cada fila. Estos sistemas surgen de áreas de aplicación muy diversas, y presentan una estructura y propiedades numéricas muy diferentes entre sí; en [55] y en sus referencias puede encontrarse una descripción más detallada de estos sistemas. El vector de términos independientes se ha construido artificialmente para que la solución exacta del sistema $Ax = b$ sea el vector unidad. De este modo es posible calcular el error de la solución calculada.

Código	Nombre	Área/Descripción	n	$nnz(A)$	$nnz(A)/n$
UF1	Pres_Poisson	Dinámica de fluidos computacional	14.822	715.804	48,29
UF2	jnlbrng1	Problema cuadrático de cojinetes	40.000	199.200	4,98
UF3	vanbody	Problema estructural	47.072	2.329.056	49,48
UF4	oilpan	Análisis estructural	73.752	2.148.558	29,13
UF5	bmwcra_1	Modelado del cigüeñal de un automóvil	148.770	10.641.602	71,53
UF6	hood	Ingeniería de automoción	220.542	9.895.422	44,87
UF7	af_0_k101	Conformación de láminas métalicas	503.625	17.550.675	34,85
UF8	af_shell3	Conformación de láminas métalicas	504.855	17.562.051	34,79
UF9	parabolic_fem	Dinámica de fluidos computacional	525.825	3.674.625	6,99
UF10	apache2	Problema estructural 3D	715.176	4.817.870	6,74
UF11	ldoor	Análisis estructural	952.203	42.493.817	44,62
UF12	ecology2	Modelado del flujo genético en animales	999.999	4.995.991	5,00
UF13	thermal2	Problema térmico en estadio estacionario	1.228.045	8.580.313	6,99
UF14	G3_circuit	Problema de simulación de circuitos	1.585.478	7.660.826	4,83

Tabla 4.11: Sistemas SDP de prueba con su código, nombre, área de aplicación de origen (o descripción), orden, número de entradas no nulas, y número medio de entradas no nulas por cada fila.

Ajuste de los algoritmos y entorno de ejecución paralelos

El ajuste de los algoritmos y entorno de ejecución paralelos para resolver los sistemas de la Tabla 4.11 es equivalente al descrito en la Sección 4.3, salvo por las siguientes diferencias:

- *Estrategia para la etapa inicial de particionado.* Escogemos la estrategia `ND-HAMD-B` (en lugar de `ND-HAMD-A`) ya que, para los sistemas de la miscelánea, esta estrategia consigue reducir en mayor medida el tiempo de ejecución de *todo* el proceso de resolución.

- *Valores de κ y τ para el cálculo del precondicionador LDL^T multinivel.* La combinación de los valores de κ y τ que logra el mejor compromiso entre tiempo y memoria depende fuertemente del sistema concreto de la miscelánea. Evaluando todas las combinaciones posibles de los valores $\kappa = 3, 5, 10$ y $\tau = 5 \times 10^{-2}, 3 \times 10^{-2}, 10^{-2}, 5 \times 10^{-3}, 3 \times 10^{-3}, 10^{-3}$, se observó que una de las siguientes tres combinaciones, $\kappa = 5$ y $\tau = 10^{-2}$, 3×10^{-3}, 10^{-3}, logra el mejor compromiso entre memoria y tiempo para muchos de los sistemas considerados en la Tabla 4.11, e incluso un compromiso razonable (en relación al mejor) cuando no lo logra. Por lo tanto, para simplificar la presentación, sólo se muestran los

resultados experimentales obtenidos al utilizar las tres combinaciones mencionadas con anterioridad.

- *Criterio de parada del método PCG.* El proceso iterativo se detiene cuando $\|x-x^{(i)}\|_A/\|x-x^{(0)}\|_A \lessapprox \epsilon$, con $\epsilon = 10^{-8}$, *o si excede las 1000 iteraciones.* En este último caso, se anota un fallo del resolutor para la combinación de los valores de κ y τ correspondiente, descartando las mediciones a la hora de presentar los resultados experimentales.

Combinaciones óptimas de τ y f para los sistemas de la miscelánea

El primer experimento que se recoge en esta sección determina las combinaciones de $\kappa = 5$, τ y f que, para cada sistema de la miscelánea y número de procesadores, logran el menor tiempo de las fases numéricas del resolutor (es decir, la suma de los tiempos de ejecución del cálculo del precondicionador LDL^T multinivel y de la resolución iterativa del sistema), la menor memoria requerida para almacenar el precondicionador (es decir, su número de entradas no nulas) y el mejor compromiso entre tiempo y memoria. Para cuantificar este compromiso utilizamos el producto de ambas magnitudes, referido como Producto Tiempo Memoria (PTM). En general, para cada sistema y valor de κ, existe habitualmente un rango de valores de τ para el cual, al aumentar la memoria requerida para almacenar el precondicionador, se reduce el tiempo de las fases numéricas del resolutor. Nos referiremos a este rango como "zona de compromiso" del resolutor. En esta zona, el PTM crece si el aumento relativo de la memoria es superior a la disminución relativa del tiempo, y decrece en caso contrario. Por lo tanto, el PTM es útil para determinar hasta qué grado la memoria adicional compensa la reducción del tiempo de ejecución de las fases numéricas del resolutor. En el caso paralelo, el valor de f no afecta de forma significativa a la memoria necesaria para almacenar el precondicionador (es decir, el número de entradas no nulas del precondicionador sólo varía ligeramente con f), aunque sí influye sobre el tiempo, de ahí que el parámetro f también sea sometido a estudio.

En las Tablas 4.12 y 4.13 se muestra, para los sistemas de la Tabla 4.11, y $p = 1, 2, 4, 8$ y 16 procesadores (valores entre paréntesis), las combinaciones de τ y f que, para $\kappa = 5$, resultaron en menor tiempo, menor memoria y menor PTM. El experimento explora, para cada sistema y número de procesadores, tres valores de τ en el intervalo $[10^{-2}, 10^{-3}]$, y cinco valores de f en el intervalo $[p, 2p]$. En particular, $\tau = 10^{-2}$, 3×10^{-3} y 10^{-3}, y $f = 1,00 \times p, 1,25 \times p, 1,50 \times p, 1,75 \times p$, y $2,00 \times p$. Las combinaciones de estos parámetros para las cuales el método PCG no convergió se descartaron. Las mediciones se realizaron conforme a la siguiente estrategia. Para cada combinación de sistema, número de procesadores, τ y f, se ejecutó diez veces la terna formada por las tres etapas del resolutor paralelo. En cada una de estas ejecuciones, se registró el tiempo de ejecución total (es decir, la suma de los tiempos de ejecución de las tres etapas), constituyendo una muestra con 10 resultados. Finalmente, se determina la ejecución correspondiente a la mediana de la muestra, considerando únicamente las mediciones tomadas en dicha ejecución.

Como se puede observar en las Tablas 4.12 y 4.13, $\tau = 10^{-2}$ fue la opción óptima en términos de memoria para *todos los sistemas de la miscelánea* excepto UF3, UF4 y UF7. Con este valor de la tolerancia de descarte, la factorización LDL^T incompleta con pivotamiento basado en la inversa descarta un mayor número de entradas del factor $\tilde{L}$, construyendo precondicionadores con mayor grado de dispersidad. Para los sistemas UF3, UF4 y UF7, $\tau = 3 \times 10^{-3}$ o $\tau = 10^{-3}$ fueron las mejores opciones en función del número de procesadores; por ejemplo, para UF4, $\tau = 3 \times 10^{-3}$ fue la opción ganadora con $p = 1$, 2 y 4 procesadores, mientras que $\tau = 10^{-3}$ lo fue con $p = 8$ y 16 procesadores. Para estos tres sistemas, $\tau = 10^{-2}$ no fue la opción óptima porque el método PCG no convergió. Además, $\tau = 3 \times 10^{-3}$ o $\tau = 10^{-3}$ fueron las mejores opciones en función del número de procesadores porque, con $\tau = 3 \times 10^{-3}$, el método PCG no convergió para el sistema UF3 y $p = 1$, 4, 8 y 16 procesadores, para el sistema UF4 y $p = 8$ y 16 procesadores, y para el sistema UF7 y cualquier número de procesadores.

Cód.	Ganador Tiempo	Ganador Memoria	Ganador PTM
UF1	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2) $\tau = 10^{-2}$ $f = 1,75 \times p$ (4) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (16) $\tau = 3 \times 10^{-3}$ $f = 1,50 \times p$ (8)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 8) $\tau = 10^{-2}$ $f = 1,25 \times p$ (4) $\tau = 10^{-2}$ $f = 1,75 \times p$ (16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 16) $\tau = 10^{-2}$ $f = 1,50 \times p$ (8) $\tau = 10^{-2}$ $f = 1,75 \times p$ (4)
UF2	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4 8) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2 4 8 16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4 8 16)
UF3	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 2) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (8)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4) $\tau = 10^{-3}$ $f = 1,75 \times p$ (8 16) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (2)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (8) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (2)
UF4	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 2 4 8 16)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (8 16) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (1 2 4)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (2 4 8 16) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (1)
UF5	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,50 \times p$ (16) $\tau = 3 \times 10^{-3}$ $f = 2,00 \times p$ (2 4 8)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 4 8 16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,25 \times p$ (8) $\tau = 10^{-2}$ $f = 1,50 \times p$ (16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (2 4)
UF6	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 2) $\tau = 10^{-3}$ $f = 1,25 \times p$ (8) $\tau = 10^{-3}$ $f = 1,75 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (4)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4 8 16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2) $\tau = 10^{-2}$ $f = 1,25 \times p$ (8) $\tau = 10^{-2}$ $f = 1,50 \times p$ (16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (4)
UF7	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 4) $\tau = 10^{-3}$ $f = 1,25 \times p$ (16) $\tau = 10^{-3}$ $f = 1,50 \times p$ (8) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (2 8 16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (4)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 4) $\tau = 10^{-3}$ $f = 1,25 \times p$ (16) $\tau = 10^{-3}$ $f = 1,50 \times p$ (8) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2)
UF8	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,75 \times p$ (8) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2 4) $\tau = 3 \times 10^{-3}$ $f = 1,50 \times p$ (16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 8 16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (2 4)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,50 \times p$ (4 16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (2 8)

Tabla 4.12: Combinaciones de τ y f que, para $\kappa = 5$, resultaron en menor tiempo, memoria y Producto Tiempo Memoria (PTM) para cada sistema y número de procesadores (valores entre paréntesis). Los resultados se refieren a los ocho primeros sistemas de la Tabla 4.11.

Cód.	Ganador Tiempo	Ganador Memoria	Ganador PTM
UF9	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4 8 16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4) $\tau = 10^{-2}$ $f = 1,25 \times p$ (8 16)	$\tau = 10^{-2}$ $f = 1,25 \times p$ (16) $\tau = 10^{-2}$ $f = 1,50 \times p$ (8) $\tau = 10^{-2}$ $f = 1,75 \times p$ (4) $\tau = 10^{-2}$ $f = 2,00 \times p$ (2) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (1)
UF10	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 3 \times 10^{-3}$ $f = 1,25 \times p$ (4 8) $\tau = 3 \times 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 3 \times 10^{-3}$ $f = 1,75 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 4 8 16) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,25 \times p$ (8 16) $\tau = 10^{-2}$ $f = 1,75 \times p$ (2 4)
UF11	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1 2) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (8)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 8 16) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2 4)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2) $\tau = 10^{-2}$ $f = 1,50 \times p$ (8 16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (4)
UF12	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4 8) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4 8 16)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4 8) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2) $\tau = 3 \times 10^{-3}$ $f = 1,50 \times p$ (16)
UF13	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,25 \times p$ (8) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2 4)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 2 4 8) $\tau = 10^{-2}$ $f = 1,75 \times p$ (16)	$\tau = 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 10^{-3}$ $f = 1,50 \times p$ (16) $\tau = 10^{-3}$ $f = 2,00 \times p$ (2) $\tau = 3 \times 10^{-3}$ $f = 1,25 \times p$ (8) $\tau = 3 \times 10^{-3}$ $f = 2,00 \times p$ (4)
UF14	$\tau = 10^{-3}$ $f = 1,00 \times p$ (2) $\tau = 10^{-3}$ $f = 1,25 \times p$ (4) $\tau = 10^{-3}$ $f = 2,00 \times p$ (8) $\tau = 3 \times 10^{-3}$ $f = 1,00 \times p$ (1) $\tau = 3 \times 10^{-3}$ $f = 1,25 \times p$ (16)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1 4 8 16) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2)	$\tau = 10^{-2}$ $f = 1,00 \times p$ (1) $\tau = 10^{-2}$ $f = 1,25 \times p$ (2 16) $\tau = 10^{-2}$ $f = 2,00 \times p$ (4 8)

Tabla 4.13: Combinaciones de τ y f que, para $\kappa = 5$, resultaron en menor tiempo, memoria y Producto Tiempo Memoria (PTM) para cada sistema y número de procesadores (valores entre paréntesis). Los resultados se refieren a los últimos seis sistemas de la Tabla 4.11.

Por otra parte, en lo que respecta al tiempo de las fases numéricas del resolutor, podemos observar que $\tau = 10^{-3}$ fue la opción ganadora para los sistemas UF3, UF4, UF6, UF7, UF9, UF11, UF12, y UF13. Para los sistemas UF1, UF2 y UF5, las mejores opciones fueron $\tau = 10^{-2}$ o $\tau = 3 \times 10^{-3}$, dependiendo del número de procesadores; por ejemplo, $\tau = 10^{-2}$ fue la opción ganadora para UF2 y $p = 1$, 2, 4 y 8 procesadores, mientras que $\tau = 3 \times 10^{-3}$ lo fue para $p = 16$ procesadores. Para estos tres sistemas, el tiempo del resolutor secuencial de ILUPACK es similar con $\tau = 10^{-2}$ y $\tau = 3 \times 10^{-3}$. Al aumentar el número de procesadores, el rendimiento relativo del resolutor en función de τ cambia en función del grado de eficiencia alcanzado por el enfoque de paralelización para cada uno de estos valores, justificando por qué la opción óptima depende del número de procesadores. Lo mismo ocurre con los sistemas UF8, UF10 y UF14, aunque en este caso las opciones ganadoras en función del número de procesadores fueron $\tau = 3 \times 10^{-3}$ o $\tau = 10^{-3}$. Por último, con respecto al parámetro f, podemos observar, por una parte, que la opción que resulta en el menor tiempo de ejecución depende fuertemente del sistema y número de procesadores, y, por otra, que para muchas combinaciones de sistema y número de procesadores, valores de $f \geq 1,50 \times p$ resultan en el menor tiempo. Esta última observación pone de manifiesto que el sobrecoste asociado a la extracción de un mayor número de tareas se puede amortizar en muchos casos con la mejora del equilibrio de carga que supone la aplicación de estrategias dinámicas de planificación.

En lo que respecta al PTM, $\tau = 10^{-2}$ fue la mejor opción para los sistemas UF1, UF2, UF5, UF6, UF8, UF9, UF10 y UF14. Por lo tanto, para los sistemas UF8, UF9, UF10 y UF14, el tamaño adicional del precondicionador en memoria con $\tau = 3 \times 10^{-3}$ o $\tau = 10^{-3}$ (con respecto a $\tau = 10^{-2}$), no compensó la reducción del tiempo de ejecución. Sin embargo, este tamaño adicional sí que compensó en el caso de los sistemas UF12 y UF13, y UF3, UF4 y UF7, aunque para este último grupo porque el método PCG no convergió con $\tau = 10^{-2}$.

Como conclusión final del experimento, cabe remarcar que los resultados recogidos en las Tablas 4.12 y 4.13 revelan que, para $\kappa = 5$, y los sistemas de la miscelánea, el intervalo $[10^{-2}, 10^{-3}]$ de valores para τ se encuentra comprendido dentro de la zona de compromiso del resolutor de ILUPACK.

Desempeño del cálculo paralelo del precondicionador y resolución iterativa del sistema

El siguiente experimento compara el desempeño del resolutor secuencial incluido en la biblioteca ILUPACK con el de nuestro resolutor paralelo para resolver los sistemas de la miscelánea. Según lo especificado en las Tablas 4.14 y 4.15, el valor de τ se puede ajustar para cada sistema con el propósito de minimizar la memoria que ocupa el precondicionador en memoria, el tiempo de las fases numéricas del resolutor y/o el PTM. Para simplificar la presentación, en este subapartado se toma $\kappa = 5$ y $\tau = 10^{-3}$ para la resolución de todos los sistemas, aunque es necesario remarcar que el grado de aceleración obtenido por el enfoque de paralelización también es una función del valor de estos parámetros. La altura (número de niveles) y el número de hojas del árbol de tareas empleado por el resolutor paralelo se fijaron durante la etapa inicial de particionado como una función del parámetro f (ver Algoritmo 2.20). El experimento explora diferentes valores de f en el intervalo $[p, 2p]$. Para simplificar la presentación, únicamente se muestran los resultados obtenidos con un *único* valor de f en el intervalo $[p, 2p]$, en particular aquel que redujo en mayor medida la suma de los tiempos de ejecución del cálculo del precondicionador multinivel y de la resolución iterativa del sistema; en las Tablas 4.12 y 4.13 se muestran los valores de f que resultaron en menor tiempo para cada número de procesadores en el caso de los sistemas UF3, UF4, UF6, UF7, UF9, UF11, UF12, y UF13 (es decir, aquellos para los cuales $\tau = 10^{-3}$ fue la opción ganadora en tiempo).

En las Tablas 4.14 y 4.15 se compara el rendimiento del resolutor secuencial incluido en la biblioteca ILUPACK (resultados para $p = 1$ en las tablas) con el de nuestro resolutor paralelo utilizando $p = 2$, 4, 8 o 16 procesadores para precondicionar y resolver iterativamente en paralelo los sistemas de la Tabla 4.1. En concreto, la Tabla 4.14 presenta los resultados para los ocho

primeros sistemas, y la Tabla 4.15, para los seis últimos. Los datos de la columna etiquetada como "Total" muestran el tiempo de ejecución total, y los situados debajo de la etiqueta "Disecc. anid." el tiempo de la disección anidada. La ejecución de esta primera etapa es completamente secuencial, de ahí que el tiempo de ejecución de la disección anidada aumente con p para todos los sistemas.

Las columnas etiquetadas como "Precond. LDL^T Multiniv." en las Tablas 4.14, y 4.15 muestran el rendimiento del cálculo paralelo del precondicionador LDL^T multinivel, y su significado coincide con el de las columnas homólogas de las Tablas 4.2, 4.3 y 4.4. En primer lugar, podemos observar que *la aceleración obtenida por nuestro enfoque aumenta con p para todos los sistemas de la miscelánea*. Por ejemplo, en el caso del sistema UF1, el enfoque de paralelización obtiene aceleraciones de 1,9, 3,5, 5,8 y 8,4, para $p = 2$, 4, 8 y 16 procesadores, respectivamente, y de 2,0, 3,7, 7,5, 13,6 para el sistema UF7 y el mismo número de procesadores. En segundo lugar, el ritmo al que aumenta la aceleración con p depende fuertemente del sistema concreto de la miscelánea. El presente experimento no pretende estudiar detalladamente el rendimiento obtenido por nuestro enfoque para cada sistema, sino demostrar que también es capaz de obtener aceleraciones notables para matrices irregulares, con estructura y propiedades numéricas muy diferentes entre sí. Por último, podemos observar que el número de elementos no nulos del precondicionador sólo varía ligeramente con p para todos los sistemas, resultado que está en concordancia con el observado en la resolución numérica de la EDP tridimensional (4.1).

Las columnas etiquetadas como "Método PCG" en las Tablas 4.14, y 4.15 ilustran el rendimiento de la resolución iterativa del sistema. Como se puede observar en estas tablas, el número de iteraciones requeridas por el método PCG paralelo para converger a la solución del sistema aumentó (moderadamente) con p (excepto para el sistema UF3, que aumentó o disminuyó con p). Este crecimiento fue más moderado para los sistemas UF12 (3 %) , UF13 (5 %), y UF14 (10 %), y más evidente para los sistemas UF1 (28 %), UF4 (35 %) y UF6 (83 %). Creemos que la degradación de la velocidad de convergencia del método con p se debe a la irregularidad subyacente a los sistemas de la miscelánea, que causa, hasta cierto grado, una distribución diferente de las incógnitas entre los niveles algebraicos de la jerarquía construida por el método paralelo al aumentar el número de procesadores. A pesar de que esta degradación reduce la eficiencia paralela del enfoque para esta etapa (comparar, por ejemplo, la aceleración del cálculo del precondicionador y de la resolución iterativa para el sistema UF6), el incremento en el número de iteraciones se puede amortizar sobradamente con el aumento de la capacidad computacional asociada a un mayor número de procesadores. De hecho, *la aceleración obtenida por nuestro enfoque para la resolución iterativa del sistema aumenta con p para todos los sistemas de la miscelánea*. En lo que respecta a la precisión de la solución calculada por el método PCG paralelo, podemos observar que sólo varía ligeramente con p y, además, que es cercana a la precisión de la solución calculada por el método secuencial (ambas afirmaciones para todos los sistemas de la miscelánea excepto UF4).

Impacto de la planificación dinámica de tareas para los sistemas de la miscelánea

El objetivo del último experimento de esta sección es cuantificar el beneficio que supone el empleo de la estrategia de planificación dinámica del Algoritmo 2.21 sobre las prestaciones paralelas (*speed-up*) del cálculo del precondicionador y la resolución iterativa del sistema. Para ello, se evalúa el rendimiento de estas dos etapas utilizando dos alternativas para ajustar el número de niveles/hojas del árbol de tareas en la etapa de particionado. La primera, a la que nos referiremos como "$f = c$", obtiene árboles de tareas de altura $log_2(p) + 1$ *completos*, es decir, con p hojas. La segunda obtiene árboles de tareas mediante el Algoritmo 2.20 (con valores de f en el intervalo $[p, 2p]$). Esta segunda alternativa produce árboles de tareas con más tareas hoja que procesadores, aumentando la probabilidad de que la estrategia de planificación dinámica del Algoritmo 2.21 mejore el equilibrio de la carga computacional. Las Figuras 4.14 y 4.15 comparan, para $p = 2$ (curvas en rojo), 4 (curvas en azul), 8 (curvas en negro) y 16

(curvas en verde) procesadores, y los catorce sistemas de la miscelánea (identificados por el valor correspondiente del eje de abscisas), la aceleración obtenida en el cálculo del precondicionador y en la resolución iterativa del sistema utilizando $f = c$ (curvas discontinuas) o f en el intervalo $[p, 2p]$ (curvas continuas). La Figura 4.14 presenta la comparación cuando se utiliza $\kappa = 5$ y $\tau = 10^{-2}$ para la construcción del precondicionador LDL^T multinivel, y la Figura 4.15 cuando se utiliza $\kappa = 5$ y $\tau = 10^{-3}$. La discontinuidades de las curvas de la Figura 4.14 se deben a la no convergencia del resolutor secuencial de ILUPACK con $\kappa = 5$ y $\tau = 10^{-2}$ para los sistemas UF3, UF4 y UF7. Como se puede observar en estas figuras, la estrategia de planificación dinámica mejora el rendimiento de ambas etapas para la mayoría de sistemas y número de procesadores, y en mayor grado para $p = 8$ y 16 procesadores. Por ejemplo, en el caso del sistema UF12, $p = 16$ procesadores, $\kappa = 5$ y $\tau = 10^{-2}$, la aceleración del precondicionador LDL^T multinivel aumentó desde 12,9 ($f = c$) hasta 15,4 ($f \in [p, 2p]$), y desde 12,1 hasta 14,2 para $\kappa = 5$ y $\tau = 10^{-3}$. Estas mejoras se pueden atribuir al mayor grado de homegenidad en la distribución de la carga logrado por la estrategia de planificación dinámica, y a las diferencias algorítmicas (diferente orden de eliminación y distribución de las incógnitas entre los niveles algebraicos) causadas por el aumento en el número de niveles/hojas del árbol de tareas.

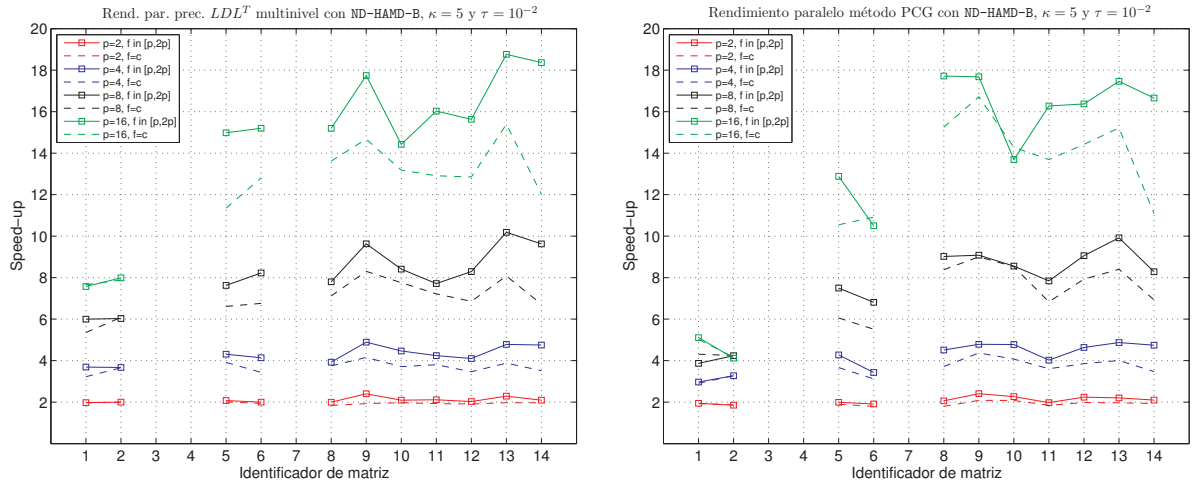


Figura 4.14: Rendimiento paralelo del cálculo del precondicionador LDL^T multinivel y del método PCG en la resolución de los sistemas de la Tabla 4.11 con ND-HAMD-B, $\kappa = 5$ y $\tau = 10^{-2}$. La discontinuidades de las curvas de la figura se deben a la no convergencia del resolutor secuencial de ILUPACK para esta combinación de parámetros.

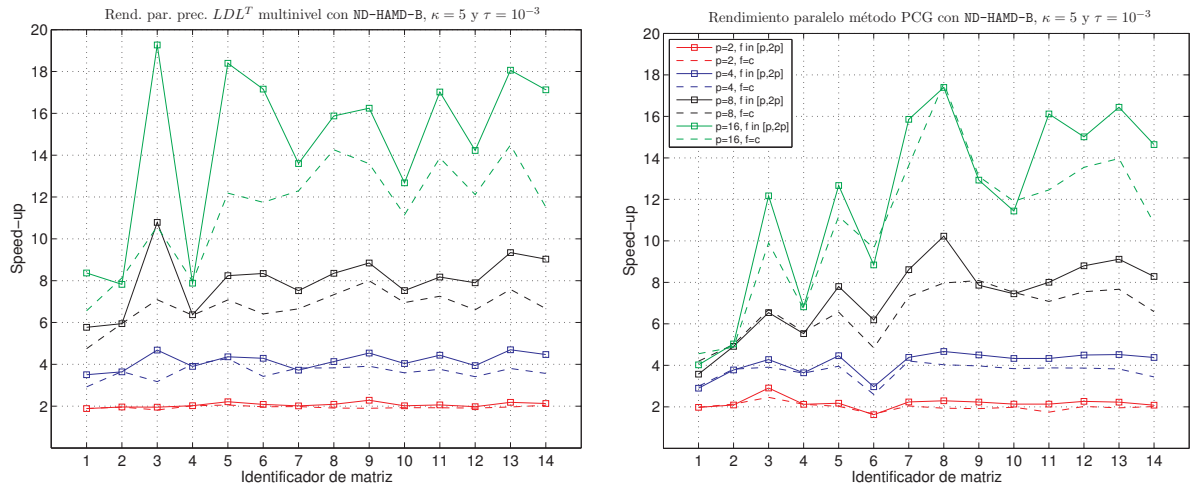


Figura 4.15: Rendimiento paralelo del cálculo del precondicionador LDL^T multinivel y del método PCG en la resolución de los sistemas de la Tabla 4.11 con ND-HAMD-B, $\kappa = 5$ y $\tau = 10^{-3}$.

Cód.	p	Total	Disecc. Anid.		Precond. LDL^T Multiniv.			Método PCG			
		T_p (seg.)	Opc. ND-HAMD-B	T_p (seg.)	nnz_M (mill.)	T_p (seg.)	S_p (T_1/T_p)	#Iter.	Err.	T_p (seg.)	S_p (T_1/T_p)
UF1	1	2,62	B	0,00	1,0	1,81	1,0	29	6,4e-13	0,81	1,0
	2	1,43	B	0,06	1,0	0,96	1,9	30	1,5e-12	0,41	2,0
	4	0,97	B	0,17	1,0	0,52	3,5	33	3,0e-12	0,28	2,9
	8	0,79	B	0,25	1,1	0,31	5,8	36	8,3e-13	0,23	3,6
	16	0,77	B	0,35	1,1	0,22	8,4	37	1,9e-12	0,20	4,0
UF2	1	0,56	B	0,00	0,3	0,41	1,0	10	6,9e-16	0,14	1,0
	2	0,32	B	0,04	0,3	0,21	2,0	10	7,2e-16	0,07	2,1
	4	0,22	B	0,07	0,3	0,11	3,6	10	7,1e-16	0,04	3,8
	8	0,20	B	0,10	0,3	0,07	5,9	11	7,5e-16	0,03	4,9
	16	0,22	B	0,14	0,3	0,05	7,8	11	7,2e-16	0,03	5,0
UF3	1	98,71	B	0,00	3,9	12,46	1,0	615	1,2e-02	86,25	1,0
	2	36,32	B	0,26	4,0	6,38	2,0	497	2,6e-02	29,68	2,9
	4	23,30	B	0,49	3,9	2,66	4,7	705	8,6e-03	20,15	4,3
	8	15,08	B	0,74	3,8	1,16	10,8	718	2,2e-02	13,18	6,5
	16	8,59	B	0,86	3,8	0,65	19,3	627	3,2e-02	7,09	12,2
UF4	1	65,60	B	0,00	3,1	7,74	1,0	470	6,3e-07	57,86	1,0
	2	31,36	B	0,28	3,1	3,82	2,0	493	2,4e-05	27,26	2,1
	4	18,41	B	0,53	3,1	1,98	3,9	572	3,4e-04	15,89	3,6
	8	12,46	B	0,79	3,1	1,21	6,4	604	7,2e-04	10,46	5,5
	16	10,53	B	1,07	3,2	0,98	7,9	672	3,5e-04	8,48	6,8
UF5	1	432,67	B	0,00	26,7	143,70	1,0	295	1,7e-08	288,97	1,0
	2	199,86	B	1,93	26,6	65,01	2,2	313	2,0e-08	132,92	2,2
	4	100,84	B	3,15	28,2	32,95	4,4	330	2,2e-08	64,74	4,5
	8	58,19	B	3,71	29,1	17,44	8,2	338	2,2e-08	37,04	7,8
	16	35,47	B	4,84	28,5	7,81	18,4	342	3,1e-08	22,81	12,7
UF6	1	95,34	B	0,00	14,3	38,66	1,0	95	2,2e-07	56,68	1,0
	2	54,64	B	1,13	14,3	18,54	2,1	125	1,6e-07	34,96	1,6
	4	30,79	B	2,67	14,5	9,01	4,3	172	2,5e-07	19,10	3,0
	8	16,79	B	3,00	14,4	4,63	8,3	160	2,5e-07	9,15	6,2
	16	12,43	B	3,77	14,4	2,25	17,2	174	2,4e-07	6,41	8,8
UF7	1	569,11	B	0,00	17,7	29,32	1,0	636	9,0e-05	539,78	1,0
	2	259,17	B	3,35	17,8	14,55	2,0	637	9,2e-05	241,27	2,2
	4	134,32	B	3,35	17,8	7,88	3,7	637	9,2e-05	123,09	4,4
	8	72,00	B	5,45	17,7	3,90	7,5	697	8,7e-05	62,65	8,6
	16	43,60	B	7,40	17,7	2,16	13,6	697	7,7e-05	34,04	15,9
UF8	1	277,91	B	0,00	34,9	94,26	1,0	129	7,5e-08	183,65	1,0
	2	128,58	B	3,41	34,7	45,09	2,1	127	1,6e-07	80,07	2,3
	4	66,93	B	4,78	34,6	22,79	4,1	135	6,6e-08	39,35	4,7
	8	34,67	B	5,44	34,7	11,28	8,4	127	1,3e-07	17,96	10,2
	16	23,29	B	6,79	35,0	5,94	15,9	149	5,5e-08	10,56	17,4

Tabla 4.14: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución paralela de los sistemas de la Tabla 4.11. Para la etapa inicial de particionado se utilizó la estrategia `ND-HAMD-B`, y para el cálculo del precondicionador LDL^T multinivel, $\kappa = 5$ y $\tau = 10^{-3}$. Los resultados de la tabla se refieren a los ocho primeros sistemas de la Tabla 4.11.

Cód.	p	Total T_p (seg.)	Disecc. Anid. Opc. ND-HAMD-B	Disecc. Anid. T_p (seg.)	Precond. LDL^T Multiniv. nnz_M (mill.)	Precond. LDL^T Multiniv. T_p (seg.)	Precond. LDL^T Multiniv. S_p (T_1/T_p)	Método PCG #Iter.	Método PCG Err.	Método PCG T_p (seg.)	Método PCG S_p (T_1/T_p)
UF9	1	46,05	B	0,00	7,7	16,54	1,0	69	1,7e-09	29,50	1,0
	2	21,77	B	1,31	7,7	7,25	2,3	74	1,9e-09	13,21	2,2
	4	11,70	B	1,51	7,7	3,64	4,5	73	2,0e-09	6,55	4,5
	8	7,61	B	1,98	7,8	1,87	8,8	77	1,5e-09	3,75	7,9
	16	5,74	B	2,44	7,8	1,02	16,2	82	1,0e-09	2,28	12,9
UF10	1	118,63	B	0,00	17,2	43,23	1,0	99	3,0e-10	75,39	1,0
	2	58,84	B	2,01	17,4	21,44	2,0	105	2,5e-10	35,39	2,1
	4	31,56	B	3,45	17,7	10,70	4,0	104	1,0e-09	17,41	4,3
	8	20,21	B	4,34	17,8	5,74	7,5	110	1,6e-10	10,12	7,4
	16	15,71	B	5,71	18,2	3,41	12,7	114	1,4e-10	6,59	11,4
UF11	1	597,56	B	0,00	61,2	166,47	1,0	151	1,1e-06	431,08	1,0
	2	287,88	B	4,93	61,0	80,74	2,1	155	4,9e-07	202,21	2,1
	4	146,64	B	9,59	60,9	37,54	4,4	167	1,2e-06	99,51	4,3
	8	89,27	B	15,06	60,5	20,37	8,2	192	9,8e-07	53,85	8,0
	16	52,59	B	16,07	60,6	9,78	17,0	193	9,8e-07	26,75	16,1
UF12	1	119,53	B	0,00	12,1	24,19	1,0	133	1,1e-10	95,32	1,0
	2	56,43	B	2,10	12,5	12,25	2,0	132	9,9e-11	42,07	2,3
	4	29,77	B	2,44	12,6	6,14	3,9	137	8,1e-11	21,19	4,5
	8	17,10	B	3,20	12,7	3,06	7,9	137	1,2e-10	10,83	8,8
	16	12,02	B	3,97	12,9	1,70	14,2	137	2,3e-10	6,35	15,0
UF13	1	183,31	B	0,00	18,3	40,19	1,0	134	2,3e-08	143,11	1,0
	2	85,53	B	2,83	18,4	18,41	2,2	137	1,6e-08	64,29	2,2
	4	44,18	B	3,97	18,5	8,56	4,7	142	2,0e-08	31,66	4,5
	8	24,43	B	4,43	18,5	4,30	9,3	141	2,3e-08	15,70	9,1
	16	16,40	B	5,47	18,5	2,22	18,1	141	1,5e-08	8,71	16,4
UF14	1	127,68	B	0,00	23,4	58,92	1,0	54	1,4e-06	68,74	1,0
	2	62,84	B	2,09	23,5	27,73	2,1	55	1,0e-06	33,03	2,1
	4	33,13	B	4,24	23,5	13,19	4,5	56	8,9e-07	15,70	4,4
	8	21,26	B	6,43	23,7	6,53	9,0	59	1,4e-06	8,30	8,3
	16	15,21	B	7,08	23,7	3,44	17,1	60	5,8e-07	4,69	14,6

Tabla 4.15: Rendimiento de las tres etapas (disección anidada multinivel, cálculo del precondicionador LDL^T multinivel y resolución del sistema precondicionado) involucradas en la resolución paralela de los sistemas de la Tabla 4.11. Para la etapa inicial de particionado se utilizó la estrategia `ND-HAMD-B`, y para el cálculo del precondicionador LDL^T multinivel, $\kappa = 5$ y $\tau = 10^{-3}$. Los resultados se refieren a los últimos seis sistemas de la Tabla 4.11.

Capítulo 5

Conclusiones y líneas abiertas de investigación

En este capítulo se discuten las conclusiones y aportaciones más relevantes del trabajo realizado junto con una serie de líneas abiertas de investigación. La estructura del capítulo es la siguiente. En la Sección 5.1 presentamos las conclusiones y aportaciones de la tesis. A continuación, en la Sección 5.2, enumeramos las publicaciones derivadas del trabajo realizado y, por último, en la Sección 5.3 presentamos una serie de problemas no resueltos hasta la fecha, que constituyen líneas abiertas de investigación.

5.1. Conclusiones y aportaciones

La resolución de sistemas de ecuaciones lineales dispersos y de gran dimensión es uno de los problemas del álgebra lineal moderna que surge con mayor frecuencia en aplicaciones de ciencia e ingeniería. La incesante demanda de mayor precisión y realismo en las simulaciones requiere el uso de modelos computacionales tridimensionales cada vez más elaborados, lo que se traduce en un aumento del tamaño y complejidad de los sistemas y del tiempo de simulación. La resolución de estos sistemas en un tiempo razonable requiere algoritmos con un alto grado de eficiencia y escalabilidad algorítmica, es decir, métodos cuyas demandas computacionales y de memoria sólo crezcan moderadamente con el tamaño del sistema, y arquitecturas de computadores paralelas que dispongan de los suficientes recursos computacionales. En los últimos años se han producido avances importantes en el ámbito de la computación de altas prestaciones, y más concretamente, en el desarrollo de procesadores con múltiples núcleos (procesadores *multicore* y *manycore*). Estas arquitecturas abren nuevas perspectivas y desafíos, como el desarrollo de algoritmos y software paralelos de resolución de sistemas lineales capaces de aprovechar de forma eficiente el paralelismo disponible explícitamente en el hardware, por ejemplo, en forma de múltiples núcleos de procesamiento.

El trabajo realizado ha afrontado el análisis, desarrollo e implementación de algoritmos paralelos capaces de identificar, extraer y aprovechar eficientemente el paralelismo disponible en el precondicionado y la resolución iterativa de sistemas lineales dispersos y de gran dimensión sobre los multiprocesadores de memoria compartida. En particular, siguiendo la línea marcada por el objetivo general de la tesis, este trabajo demuestra experimentalmente, para los problemas de aplicación evaluados, que el grado de paralelismo de tareas presente en los métodos numéricos de ILUPACK es suficiente para la ejecución eficiente de implementaciones paralelas de estos métodos sobre multiprocesadores de memoria compartida. A continuación, enumeramos las contribuciones que son aportaciones propias y originales:

- Desarrollo de un nuevo algoritmo intrínsecamente paralelo para la construcción de precondicionadores multinivel basados en la factorización LDL^T con pivotamiento basado en la

inversa. El paralelismo intrínseco en este método se pone de manifiesto mediante la combinación de dos ingredientes básicos; en primer lugar, el sistema se reordena conforme a la jerarquía multinivel de subgrafos independientes y separadores construida por la disección anidada; en segundo lugar, durante la computación numérica, la jerarquía multinivel construida por ILUPACK se entrelaza *estratégicamente* con la jerarquía construida por el proceso recursivo de disección. El algoritmo resultante acomoda la semántica del enfoque de precondicionado de la biblioteca ILUPACK, mientras expone simultáneamente un alto grado de concurrencia.

Ninguno de los métodos disponibles hasta la fecha acomoda la semántica de los precondicionadores basados en la factorización ILU multinivel de ILUPACK mientras expone simultáneamente un alto grado de concurrencia. El único enfoque que guarda cierta conexión con el método propuesto es el de la biblioteca pARMS, aunque presenta algunas diferencias estratégicas. En primer lugar, la estrategia de selección de malla gruesa que constituye la base del enfoque algebraico de precondicionado de pARMS se basa en la identificación de grupos de incógnitas independientes (*group independent sets*) [113, 140], en lugar de controlar el crecimiento de la norma de la inversa del factor aproximado; en segundo lugar, pARMS expone paralelismo utilizando técnicas diferentes, basadas en la descomposición de dominios [111].

- Aprovechamiento de la concurrencia intrínseca en el algoritmo anterior a nivel de tareas a partir del árbol de dependencias, y ejecución concurrente de este árbol mediante una estrategia de mapeado y planificación *dinámica* de tareas a procesos para mejorar el equilibrio de la carga computacional durante el proceso de construcción del precondicionador. El enfoque aplicado para la extracción del paralelismo de tareas acomoda la aproximación algebraica de ILUPACK en el árbol de dependencias entre tareas, y aumenta aún más el grado de concurrencia mediante el cálculo y almacenamiento, de forma local e independiente, de los datos propios de las tareas y de los datos correspondientes a sus ancestros en el árbol. Estos últimos, denominados *bloques locales de contribución*, almacenan las contribuciones locales de las tareas a las entradas del complemento de Schur aproximado correspondientes a sus ancestros en el árbol. De esta forma se evitan operaciones de reducción (sincronización) de estas contribuciones hasta que sus ancestros no requieran iniciar el proceso de factorización de sus bloques locales. Además, el enfoque también acomoda a las estructuras de datos locales de las tareas, las técnicas que se utilizan para enriquecer el enfoque de precondicionado de ILUPACK, como la aplicación de escalados y permutaciones (para la reducción de llenado) locales, y adapta a estas estructuras de datos las reglas de descarte y la técnica de Ajiz-Jennings utilizadas por la factorización LDL^T con pivotamiento basado en la inversa.

  La aproximación propuesta para aprovechar el paralelismo de tareas durante el proceso de construcción del precondicionador también se aplica para la resolución iterativa del sistema precondicionado. La estructura lógica de la variante paralela de los precondicionadores LDL^T multinivel habilita de forma directa el aprovechamiento del paralelismo de tareas durante su aplicación. Sin embargo, debido al bajo grado de reutilización de datos presente en las computaciones del proceso de resolución regresiva y progresiva, y a la organización del sistema de memoria de los multiprocesadores de memoria compartida, se ha investigado el empleo de estrategias *estáticas* de mapeado para reducir los sobrecostes asociados al movimiento de las estructuras de datos de las tareas entre los procesadores. En particular, durante la resolución progresiva, el enfoque mantiene invariablemente el mapeado de tareas a procesadores que resulta del cálculo paralelo del precondicionador y, durante la resolución regresiva, se pueden realizar dinámicamente ajustes en este mapeado con el propósito de reducir los tiempos de espera de los procesos. La paralelización del resto de las operaciones del método de los gradientes conjugados, es decir, el producto matriz dispersa

por vector, los productos escalares y las actualizaciones de vectores, se realiza de forma coordinada con la aplicación del precondicionador, con el objeto de aumentar la localidad de referencia y reducir el sobrecoste asociado al tráfico de datos en el sistema de memoria del multiprocesador. En particular, las tareas hoja almacenan y calculan, de forma local e independiente, contribuciones a/réplicas de las entradas (de los vectores) correspondientes a sus ancestros, de tal forma que el particionado de los datos coincide para todas las operaciones del cuerpo del método PCG .

Aunque la extracción automática de paralelismo de tareas, combinada con el uso de planificación dinámica de tareas a procesadores, se ha aplicado con éxito en otros trabajos para la paralelización de métodos directos de resolución de sistemas de ecuaciones lineales dispersos sobre multiprocesadores de memoria compartida [57, 142, 143, 144, 145] y distribuida [15], la aproximación desarrollada en esta tesis es original por el contexto en el que se propone: la construcción de precondicionadores LDL^T multinivel y la resolución iterativa del sistema precondicionado, y los multiprocesadores de memoria compartida como arquitectura destino. La gran mayoría de los métodos paralelos propuestos en la literatura para construir precondicionadores ILU y resolver iterativamente sistemas de ecuaciones lineales dispersos tiene como arquitectura destino los multiprocesadores de memoria distribuida y rara vez utiliza estrategias dinámicas de planificación; consultar, por ejemplo, [96, 100, 101, 111, 120].

- Implementación de los algoritmos paralelos siguiendo los principios de diseño de la biblioteca ILUPACK. Los códigos paralelos resultantes son compatibles con las estructuras de datos, gestión de memoria, prototipos y organización de las rutinas de la versión 2.1 de esta biblioteca. Los nuevas rutinas desarrolladas para el precondicionado y resolución iterativa están escritas en los lenguajes C y Fortran-77, y utilizan el estándar OpenMP, revisión 2.5, para aprovechar el modelo de programación paralela de variables compartidas. Además, mediante la definición de macros en tiempo de compilación, las rutinas paralelas pueden trabajar en aritmética en coma flotante IEEE 754 de simple o doble precisión. Los códigos se pueden aplicar para resolver sistemas de ecuaciones lineales reales (o complejos) simétricos (o hermíticos) definidos positivos arbitrarios, aunque su extensión para la resolución de sistemas de ecuaciones más generales no es trivial, y hace necesario mayores esfuerzos de investigación. Los códigos presentan un alto grado de portabilidad al estar programados utilizando herramientas estándar para el desarrollo de aplicaciones paralelas, pudiendo ser compilados y ejecutados sobre cualquier multiprocesador de memoria compartida o procesador multinúcleo de propósito general que disponga de compiladores de C y Fortran-77 con soporte para las directivas y el entorno de ejecución de OpenMP.

- Evaluación experimental detallada del rendimiento de los algoritmos paralelos numéricos desarrollados para resolver los sistemas de ecuaciones dispersos y de gran dimensión que aparecen ligados a varias EDPs bidimensionales y tridimensionales sobre un multiprocesador de memoria compartida comercial; extensión de este estudio a una miscelánea de sistemas ligados a problemas más irregulares, provenientes de diversas áreas de aplicación; y evaluación de aproximaciones paralelas actuales para el particionado de matrices dispersas. De los experimentos realizados en el marco de estos problemas de aplicación, podemos extraer las siguientes conclusiones:

  - El grado de eficiencia paralela de nuestro enfoque depende de dos factores: la distribución del coste del nuevo algoritmo paralelo entre los niveles del árbol de tareas, y el equilibrio que éste logre entre el beneficio que supone la ejecución concurrente de tareas y la penalización asociada a los factores de sobrecoste. Con respecto al primero, cabe mencionar que el grado de paralelismo será mayor cuanto mayor sea la concentración del coste en el último nivel del árbol de tareas. Del segundo, decir que

los factores de sobrecoste con mayor peso son los tiempos de espera de los procesos, causados por desequilibrios de carga, la reducción del grado de paralelismo para los niveles más cercanos a la raíz del árbol, y el acceso/cálculo por duplicado de las entradas de los bloques locales de contribución de las tareas. Ambos factores dependen del tamaño relativo de los separadores con respecto al de los subgrafos independientes, y del grado de eficiencia y escalabilidad del enfoque algebraico de precondicionado.

- El grado de eficiencia alcanzado por el enfoque es alto para resolver sistemas de ecuaciones de gran dimensión ligados a una EDP elíptica tridimesional en un multiprocesador de memoria compartida comercial. Excepto para los sistemas de ecuaciones de menor dimensión, la aceleración paralela del proceso de construcción del precondicionador y de la resolución iterativa del sistema siempre aumenta con el número de procesadores. Además, al aumentar el tamaño del sistema (para un número fijo de procesadores), la eficiencia generalmente crece, llegando a obtener aceleraciones paralelas notables para los sistemas de mayor dimensión. La evaluación de la eficiencia del enfoque para este problema de aplicación, se ha complementado con un estudio detallado del sobrecoste computacional y de memoria asociado a la extracción de paralelismo de tareas, que evidencia, por una parte, el crecimiento moderado de estos sobrecostes al aumentar el número de procesadores y, por otra, su rápida reducción al aumentar el tamaño del sistema. Este comportamiento de los sobrecostes asociados a la extracción del paralelismo de tareas, junto con la elevada concentración del coste computacional del método en los últimos niveles del árbol, justifican el alto grado de efectividad alcanzado por nuestro enfoque para resolver estos sistemas. Además, también hemos comprobado que el enfoque obtiene eficiencias razonables para sistemas ligados a problemas más irregulares, provenientes de diversas áreas de aplicación.
- La planificación dinámica de tareas a procesos permite aumentar la eficiencia paralela del enfoque siempre y cuando la mejora que supone el mayor grado de homogeneidad en la distribución de la carga compense los sobrecostes asociados a la extracción del paralelismo de tareas. Para los sistemas ligados a la EDP tridimensional considerada, esta mejora se puede amortizar sobradamente para los sistemas de tamaños intermedios y grandes.
- El mapeado de tareas a procesos que resulta del cálculo del precondicionador también permite explotar de forma eficaz la concurrencia disponible en el árbol para los procesos de resolución progresiva y regresiva. Además, los ajustes dinámicos que realiza el proceso de resolución regresiva sobre este mapeado sólo penalizan su rendimiento de forma moderada en pocos casos, y lo mejoran en muchos casos, en algunos de forma significativa, a causa de la reducción de los tiempos de espera de los procesos derivada de estos ajustes.
- El enfoque de paralelización preserva la semántica de las técnicas de precondicionado de ILUPACK. Los experimentos realizados con los sistemas ligados a EDPs bidimensionales y tridimensionales confirman un bajo grado de sensibilidad del número de elementos no nulos del precondicionador, número de iteraciones y precisión de la solución calculada, al número de procesadores involucrados en el cálculo paralelo. Estas evidencias se han complementado con otras que ponen de manifiesto la similitud de la jerarquía multinivel construida por el método secuencial de ILUPACK y la construida por nuestro algoritmo paralelo. Aunque para sistemas de ecuaciones más irregulares, sí que se observa un mayor grado de sensibilidad del rendimiento numérico del método a un aumento del número de procesadores, el incremento en el número de iteraciones se puede amortizar de sobra con el aumento de la capacidad computacional asociada a un mayor número de procesadores.
- Nuestro enfoque de paralelización obtiene eficiencias elevadas para el precondicionado y resolución iterativa del sistema con los particionados calculados por cualquiera de

las dos aproximaciones paralelas evaluadas para el cálculo de la disección anidada. El estudio realizado en el marco de la resolución de la EDP tridimensional revela, por una parte, la mayor eficiencia paralela y de memoria de ParMETIS frente a PT-SCOTCH, lo que se traduce en una reducción del tiempo de ejecución de todo el proceso y, por otra, que el rendimiento de nuestro método numérico paralelo no se degrada al aumentar el número de procesadores involucrados en el cálculo paralelo de la disección anidada de ParMETIS, evidenciando su bajo grado de sensibilidad a los compromisos que en esta biblioteca ocasionan una reducción de la calidad de los separadores. Además, según lo indicado por otros estudios [48], hemos comprobado que los métodos directos para la resolución de sistemas de ecuaciones lineales dispersos sí presentan un alto grado de sensibilidad a esta reducción de la calidad de los separadores, lo que se pone de manifiesto con un aumento significativo del número de operaciones necesarias para completar una factorización de Cholesky dispersa directa del sistema reordenado mediante la disección anidada paralela de ParMETIS.

5.2. Difusión de los resultados del trabajo

Los resultados del trabajo desarrollado en esta tesis han sido sometidos para su publicación en un artículo de revista científica, publicados en actas de conferencias internacionales y nacionales, y difundidos en ponencias de conferencias internacionales.

Revistas

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Exploiting Thread-Level Parallelism in the Iterative Solution of Sparse Linear Systems. *Parallel Computing*. En revisión (1ª revisión, abril 2010; 2ª revisión).

Actas de conferencias internacionales

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors. In *Parallel Computing: Architectures, Algorithms and Applications* (2007), C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, and F. Peters, Eds., vol. 38 of *Advances in Parallel Computing*, pp. 287–294. PARCO'07, Jülich (Alemania).

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Design, Tuning and Evaluation of Parallel Multilevel ILU Preconditioners. In *High Performance Computing for Computational Science - VECPAR 2008* (2008), J. Palma, P. Amestoy, M. Dayde, M. Mattoso, and J. C. Lopes, Eds., vol. 5336 of *Lecture Notes in Computer Science*, Springer, pp. 314–327. VECPAR'08, Toulouse (Francia).

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Scheduling Strategies for Parallel Sparse Backward/Forward Substitution. In *9-th International Workshop on State-of-the-Art in Scientific and Parallel Computing* (2008), Lecture Notes in Computer Science. PARA'08, Trondheim (Noruega). En revisión.

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Recent advances in the parallel iterative solution of large-scale sparse linear systems. In *9th International Conference on Computational and Mathematical Methods in Science and Engineering* (2009), vol. 1, pp. 68–72. CMMSE'09, Gijón (España).

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Evaluation of Parallel Sparse Matrix Partitioning Software for Parallel Multilevel ILU Preconditioning

on Shared-Memory Multiprocessors. In *Parallel Computing: From Multicores and GPU's to Petascale* (2009), B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and T. Priol, Eds., vol. 19 of *Advances in Parallel Computing*, pp. 125–132. PARCO'09, Lyon (Francia).

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallelization of Multilevel ILU Preconditioners on Distributed-Memory Multiprocessors. In *PARA 2010: State of the Art in Scientific and Parallel Computing* (2010), Lecture Notes in Computer Science. PARA'10, Reykjavík (Islandia). En revisión.

Actas de conferencias nacionales

- Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Optimización de algoritmos de álgebra lineal dispersa para MMCs mediante KOJAK y VAMPIR. In *XVIII Jornadas de Paralelismo* (2007), pp. 759–766. Zaragoza (España).

- Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Scheduling strategies for parallel sparse linear algebra computacions on multithreaded architectures. In *XIX Jornadas de Paralelismo* (2008), pp. 51–56. Castellón (España).

- Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Avances en la Resolución Paralela de Sistemas de Ecuaciones Lineales Precondicionados mediante factorizaciones ILU Multinivel. In *Congreso de Métodos Numéricos en Ingeniería* (2009), p. 338. METNUM'09. Barcelona (España).

Ponencias en conferencias internacionales

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallel Multilevel ILU Preconditioners. *5th International Workshop on Parallel Matrix Algorithms and Applications*. Junio, 2008. Neuchâtel (Suiza)

- Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallel Preconditioning based on ILUPACK for Multithreaded Architectures. *SIAM Conference on Computational Science and Engineering*. Marzo, 2009. Miami (EE.UU).

5.3. Líneas abiertas de investigación

La tesis cubre con sus objetivos la identificación, extracción y aprovechamiento del paralelismo de tareas en los métodos de ILUPACK para la resolución eficiente de sistemas de ecuaciones lineales dispersos y de gran dimensión sobre multiprocesadores de memoria compartida. En esta misma línea de trabajo pueden identificarse una serie de problemas no resueltos hasta la fecha, que constituyen líneas abiertas de investigación:

- Desarrollar algoritmos paralelos, basados en el enfoque algebraico de ILUPACK, para el precondicionado y la resolución iterativa de sistemas de ecuaciones lineales dispersos y de gran dimensión bajo el paradigma de programación basado en el paso de mensajes. El paralelismo masivo disponible en los multiprocesadores de memoria distribuida, en forma de cientos hasta miles de procesadores, hace necesario investigar tipos adicionales de paralelismo en los métodos numéricos de ILUPACK que se adapten naturalmente a este modelo de programación, y expongan simultáneamente el suficiente grado de paralelismo como para poder explotar eficientemente su enorme capacidad computacional.

- Extender nuestro enfoque para la extracción de paralelismo de tareas para desarrollar algoritmos paralelos de resolución de sistemas lineales de ecuaciones más generales, como

los casos simétrico indefinido y no simétrico pero estructuralmente simétrico. Los métodos numéricos disponibles en ILUPACK para la resolución de este tipo de sistemas presentan diferencias estratégicas frente a los que se emplean para resolver sistemas SDP. En primer lugar, el proceso de construcción del precondicionador aplica, sobre la matriz de entrada para cada nivel de la jerarquía, heurísticos de pivotamiento simétrico estático [31, 141] para aumentar el peso relativo de la diagonal principal formada por bloques de dimensión 2×2. En segundo lugar, la factorización LDL^T con pivotamiento basado en la inversa puede aceptar/rechazar, en cada paso de eliminación, elementos pivote de dimensión 1×1 o 2×2. El trabajo a desarrollar en esta línea debe investigar la adaptación de estas técnicas a las estructuras de datos locales de las tareas, y evaluar experimentalmente el rendimiento numérico y eficiencia paralela de los algoritmos paralelos desarrollados.

- Los programas paralelos desarrollados presentan un alto grado de experimentalidad y, aunque su fiabilidad y robustez ha sido comprobada para resolver sistemas de ecuaciones ligados a diversos problemas de aplicación, resulta necesario dedicar más esfuerzos para incrementar su madurez como producto de software final, especialmente en lo que se refiere a su facilidad de uso. Además, también resulta necesario su integración en la versión actual de la biblioteca ILUPACK, la 2.3, debido a los cambios estratégicos que esta versión incorpora en la gestión de la memoria dinámica.

- Desarrollar algoritmos y software paralelos para el cálculo multinivel de separadores de vértices que permitan explotar de forma más eficiente el hardware subyacente de los multiprocesadores de memoria compartida y procesadores multinúcleo. Las aproximaciones paralelas evaluadas en este trabajo para el particionado de matrices dispersas, es decir, ParMETIS y PT-SCOTCH, siguen el modelo de programación basado en el paso de mensajes y, por tanto, su arquitectura destino son los multiprocesadores de memoria distribuida. El trabajo desarrollado en esta línea permitiría, por una parte, disponer de códigos que se puedan integrar con mayor facilidad a los desarrollados en esta tesis, y, por otra, explotar de forma más eficiente las particularidades de las arquitecturas de memoria compartida.

- Aplicar los códigos desarrollados para resolver otros problemas *reales* de interés científico e ingenieril. Los métodos numéricos de ILUPACK se han aplicado con éxito para resolver problemas tridimensionales de interés científico altamente indefinidos, como el modelo de localización de Anderson [141], o la EDP de Helmholtz en medios heterogéneos [31]. El trabajo desarrollado en esta línea permitiría reducir los tiempos de respuesta de estas aplicaciones.

Bibliografía

[1] Ajiz, M. A., and Jennings, A. A Robust Incomplete Choleski-Conjugate Gradient Algorithm. *International Journal for Numerical Methods in Engineering 20*, 5 (1984), 949–966.

[2] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Exploiting Thread-Level Parallelism in the Iterative Solution of Sparse Linear Systems. *Parallel Computing*. En revisión (1ª revisión, abril 2010; 2ª revisión).

[3] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallel Multilevel ILU Preconditioners. *5th International Workshop on Parallel Matrix Algorithms and Applications*. Junio, 2008. Neuchâtel (Suiza).

[4] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallel Preconditioning based on ILUPACK for Multithreaded Architectures. *SIAM Conference on Computational Science and Engineering*. Marzo, 2009. Miami (EE.UU).

[5] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors. In *Parallel Computing: Architectures, Algorithms and Applications* (2007), C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, and F. Peters, Eds., vol. 38 of *Advances in Parallel Computing*, pp. 287–294. PARCO'07, Jülich (Alemania).

[6] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Design, Tuning and Evaluation of Parallel Multilevel ILU Preconditioners. In *High Performance Computing for Computational Science - VECPAR 2008* (2008), J. Palma, P. Amestoy, M. Dayde, M. Mattoso, and J. C. Lopes, Eds., vol. 5336 of *Lecture Notes in Computer Science*, Springer, pp. 314–327. VECPAR'08, Toulouse (Francia).

[7] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Scheduling Strategies for Parallel Sparse Backward/Forward Substitution. In *9-th International Workshop on State-of-the-Art in Scientific and Parallel Computing* (2008), Lecture Notes in Computer Science. PARA'08, Trondheim (Noruega). En revisión.

[8] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Evaluation of Parallel Sparse Matrix Partitioning Software for Parallel Multilevel ILU Preconditioning on Shared-Memory Multiprocessors. In *Parallel Computing: From Multicores and GPU's to Petascale* (2009), B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, F. Peters, and T. Priol, Eds., vol. 19 of *Advances in Parallel Computing*, pp. 125–132. PARCO'09, Lyon (Francia).

[9] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Recent advances in the parallel iterative solution of large-scale sparse linear systems. In *9th International Conference on Computational and Mathematical Methods in Science and Engineering* (2009), vol. 1, pp. 68–72. CMMSE'09, Gijón (España).

[10] Aliaga, J. I., Bollhöfer, M., Martín, A. F., and Quintana-Ortí, E. S. Parallelization of Multilevel ILU Preconditioners on Distributed-Memory Multiprocessors. In *PARA 2010: State of the Art in Scientific and Parallel Computing* (2010), Lecture Notes in Computer Science. PARA'10, Reykjavík (Islandia). En revisión.

[11] Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Optimización de algoritmos de álgebra lineal dispersa para MMCs mediante KOJAK y VAMPIR. In *XVIII Jornadas de Paralelismo* (2007), pp. 759–766. Zaragoza (España).

[12] Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Scheduling strategies for parallel sparse linear algebra computacions on multithreaded architectures. In *XIX Jornadas de Paralelismo* (2008), pp. 51–56. Castellón (España).

[13] Aliaga, J. I., Martín, A. F., and Quintana-Ortí, E. S. Avances en la Resolución Paralela de Sistemas de Ecuaciones Lineales Precondicionados mediante factorizaciones ILU Multinivel. In *Congreso de Métodos Numéricos en Ingeniería* (2009), p. 338. METNUM'09. Barcelona (España).

[14] Amestoy, P., Davis, T. A., and Duff, I. S. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl. 17*, 4 (1996), 886–905.

[15] Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. Appl. 23*, 1 (2001), 15–41.

[16] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Croz, J. D., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. *LAPACK Users' guide (third ed.)*. SIAM Publications, Philadelphia, PA, USA, 1999.

[17] Axelsson, O. *Iterative Solution Methods*. Cambridge University Press, 1984.

[18] Axelsson, O., and Kolotilina, L. Diagonally Compensated Reduction and Related Preconditioning Methods. *Linear Algebra Appl. 1*, 2 (1995), 155–177.

[19] Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc User's Manual. Tech. Rep. ANL-95/11 - Revision 3.0.0, Argonne National Laboratory, 2008.

[20] Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc Web page, 2009. `http://www.mcs.anl.gov/petsc`.

[21] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[22] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[23] Behie, G. A., and Forsyth, P. A. Comparison of Fast Iterative Methods for Symmetric Systems. *IMA Journal Numerical Analysis 3*, 1 (1983), 41–63.

[24] Benzi, M., Haws, J. C., and Tuma, M. Preconditioning Highly Indefinite and Nonsymmetric Matrices. *SIAM J. Sci. Comput. 22*, 4 (2000), 1333–1353.

[25] Benzi, M., Joubert, W., and Mateescu, G. Numerical Experiments with Parallel Orderings for ILU Preconditioners. *Electr. Trans. Num. Anal. 8* (1999), 88–114.

[26] Benzi, M., Meyer, C. D., and Tuma, M. A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method. *SIAM J. Sci. Comput. 17*, 5 (1996), 1135–1149.

[27] Benzi, M., Szyld, D. B., and van Duin, A. Orderings for Incomplete Factorization Preconditioning of Nonsymmetric Problems. *SIAM J. Sci. Comput. 20*, 5 (1999), 1652–1670.

[28] Benzi, M., and Tuma, M. A Sparse Approximate Inverse Preconditioner for Nonsymmetric Linear Systems. *SIAM J. Sci. Comput. 19*, 3 (1998), 968–994.

[29] Bollhöfer, M. A Robust ILU with Pivoting Based on Monitoring the Growth of the Inverse Factors. *Linear Algebra Appl. 338*, 1–3 (2001), 201–218.

[30] Bollhöfer, M. A Robust and Efficient ILU that Incorporates the Growth of the Inverse Triangular Factors. *SIAM J. Sci. Comput. 25*, 1 (2004), 86–103.

[31] Bollhöfer, M., Grote, M. J., and Schenk, O. Algebraic Multilevel Preconditioner for the Helmholtz Equation in Heterogeneous Media. *SIAM J. Sci. Comput. 31*, 5 (2009), 3781–3805.

[32] Bollhöfer, M., and Saad, Y. A Factored Approximate Inverse Preconditioner With Pivoting. *SIAM J. Matrix Anal. Appl. 23*, 3 (2001), 692–705.

[33] Bollhöfer, M., and Saad, Y. On the Relations between ILUs and Factored Approximate Inverses. *SIAM J. Matrix Anal. Appl. 24*, 1 (2002), 219–237.

[34] Bollhöfer, M., and Saad, Y. Multilevel Preconditioners Constructed from Inverse–Based ILUs. *SIAM J. Sci. Comput. 27*, 5 (2006), 1627–1650. Special issue on the 8–th Copper Mountain Conference on Iterative Methods.

[35] Bollhöfer, M., Saad, Y., and Schenk, O. ILUPACK volumes 2.1 — preconditioning software package. `http://www.math.tu-berlin.de/ilupack`, year = 2006, month = Jan,.

[36] Brandt, A. Multi-Level Adaptive Solution to Boundary-Value Problems. *Math. Comp. 31* (1977), 333–390.

[37] Brandt, A. Algebraic Multigrid Theory: the Symmetric Case. *Appl. Math. Comput. 19* (1986), 23–56.

[38] Brandt, A., McCormick, S., and Ruge, J. Algebraic MultiGrid (AMG) for Automatic Multigrid Solutions with Application to Geodatic Computations. Tech. rep., Institute for Computational Studies, Fort Coolins, CO, 1982.

[39] Brandt, A., McCormick, S., and Ruge, J. Algebraic MultiGrid (AMG) for Sparse Matrix Equations. In *Sparsity and Its Applications* (1984), D. Evans, Ed., Cambridge University Press, pp. 257–284.

[40] Brezina, M., Cleary, A. J., Falgout, R. D., Henson, V. E., Jones, J. E., Manteuffel, T. A., McCormick, S. F., and Ruge, J. W. Algebraic Multigrid Based on Element Interpolation (AMGe). *SIAM J. Sci. Comput. 22*, 5 (2000), 1570–1592.

[41] Brezinski, C. *Projection Methods for Systems of Equations.* North-Holland, 1997.

[42] Bridson, R., and Tang, W.-P. A Structural Diagnosis of Some IC Orderings. *SIAM J. Sci. Comput. 22*, 5 (2000), 1527–1532.

[43] Bruaset, A. M. *A Survey of Preconditioned Iterative Methods.* Longman Scientific & Technical, 1995.

[44] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonnald, J., and Menon, R. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers, 2001.

[45] Chapman, B., and van der Pas, R. *Using OpenMP. Portable Shared Memory Parallel Programming.* The MIT Press, 2007.

[46] Chartier, T., Falgout, R. D., Henson, V. E., Jones, J., Manteuffel, T., McCormick, S., Ruge, J., and Vassilevski, P. S. Spectral AMGe (ρAMGe). *SIAM J. Sci. Comput. 25*, 1 (2003), 1–26.

[47] Chevalier, C., and Pellegrini, F. Improvement of the Efficiency of Genetic Algorithms for Scalable Parallel Graph Partitioning in a Multi-Level Framework. In *Proceedings of Euro-Par 2006* (2006), vol. 4128 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 243–252.

[48] Chevalier, C., and Pellegrini, F. PT-SCOTCH: a Tool for Efficient Parallel Graph Ordering. *Parallel Comput. 34*, 6-8 (2008), 318–331.

[49] Chow, E. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners with a Priori Sparsity Patterns. *Int. J. High Perform. Comput. Appl. 15*, 1 (2001), 56–74.

[50] Chow, E., and Saad, Y. Experimental Study of ILU Preconditioners for Indefinite Matrices. *J. Comput. Appl. Math. 86*, 2 (1997), 387–414.

[51] Chow, E., and Saad, Y. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. *SIAM J. Sci. Comput. 19*, 3 (1998), 995–1023.

[52] Clift, S. S., and pai Tang, W. Weighted Graph Based Ordering Techniques for Preconditioned Conjugate Gradient Methods. *BIT 35* (1993), 30–47.

[53] Cline, A., Moler, C. B., Stewart, G., and Wilkinson, J. An Estimate for the Condition Number of a Matrix. *SIAM J. Numer. Anal. 16* (1979), 368–375.

[54] Concus, P., Golub, G., and O'Leary, D. A Generalized Conjugate Gradient Method for the Numerical Solution of Elliptic Partial Differential Equations. In *Proceedings of Sparse Matrix Conference, Sparse Matrix Computation* (1976), J. Reid, Ed., Academic Press, p. 309–332.

[55] Davis, T. A. University of Florida Sparse Matrix Collection. *NA Digest 97*, 27 (1997). `http://www.cise.ufl.edu/research/sparse`.

[56] Davis, T. A. *Direct Methods for Sparse Linear Systems.* SIAM Publications, 2006.

[57] Demmel, W., Gilbert, J. R., and Li., X. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Anal. Appl. 20*, 4 (1999), 915–952.

[58] Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw. 16*, 1 (1990), 1–17.

[59] Dongarra, J. J., Du Croz, J., Hammarling, S., and Hanson, R. J. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw. 14*, 1 (1988), 1–17.

[60] Douglas, J., and Rachford, H. H. On the Numerical Solution of Heat Conduction Problems in Two and Three Space Variables. *Transactions of the American Mathematical Society 82*, 2 (1956), 421–439.

[61] Drkovsova, J., Greenbaum, A., Rozlovznik, M., and Strakovs, Z. Numerical Stability of the GMRES Method. *BIT 35* (1995), 309–330.

[62] Duff, I. S., Erisman, A. M., and Reid, J. K. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[63] Duff, I. S., and Koster, J. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl. 20*, 4 (1999), 889–901.

[64] Duff, I. S., and Koster, J. On Algorithms for Permuting Large Entries to the Diagonal of a Sparse Matrix. *SIAM J. Matrix Anal. Appl. 22*, 4 (2000), 973–996.

[65] Duff, I. S., and Meurant, G. A. The Effect of Ordering on Preconditioned Conjugate Gradients. *BIT 29*, 4 (1989), 635–657.

[66] Dutto, L. C. The Effect of Ordering on Preconditioned GMRES Algorithm, for Solving the Compressible Navier-Stokes Equations. *International Journal for Numerical Methods in Engineering 36*, 3 (1981), 457–497.

[67] Elman, H. C. A Stability Analysis of Incomplete LU Factorizations. *Mathematics of Computation 47*, 175 (1986), 191–217.

[68] Elman, H. C. Relaxed and Stabilized Incomplete Factorizations for Non-Self-Adjoint Linear Systems. *BIT 29*, 4 (1989), 890–915.

[69] Fiduccia, C. M., and Mattheyses, R. M. A Linear-Time Heuristic for Improving Network Partitions. In *DAC '82: Proceedings of the 19th conference on Design automation* (Piscataway, NJ, USA, 1982), IEEE Press, pp. 175–181.

[70] Fischer, B. *Polynomial Based Iteration Methods for Symmetric Linear Systems*. Wiley-Teubner, 1996.

[71] Forsythe, G. E., and Straus, E. G. On Best Conditioned Matrices. *Proc. Amer. Math. Soc. 6* (1955), 1955.

[72] Freund, R., and Nachtigal, N. QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems. *Numer. Math. 60*, 315 (1991), 315–339.

[73] Gee, M. W., Siefert, C. M., Hu, J. J., Tuminaro, R. S., and Sala, M. G. ML 5.0 Smoothed Aggregation User's Guide. Tech. Rep. SAND2006-2649, 2006.

[74] George, A. Nested Dissection of a Regular Finite Element Mesh. *SIAM J. Numer. Anal. 10*, 2 (1973), 345–363.

[75] George, A., and Liu, W. H. The Evolution of the Minimum Degree Ordering Algorithm. *SIAM Review 31*, 1 (1989), 1–19.

[76] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 3rd ed. John Hopkins, Baltimore MD, 1996.

[77] Golub, G. H., and O'Leary, D. P. Some History of the Conjugate Gradient and Lanczos Methods. *SIAM Review 31*, 1 (1989), 50–102.

[78] Golub, G. H., and Ye, Q. Inexact Preconditioned Conjugate Gradient Method with Inner-Outer Iteration. *SIAM J. Sci. Comput. 21*, 4 (2000), 1305–1320.

[79] Grama, A., Gupta, A., Karypis, G., and Kumar, V. *Introduction to Parallel Computing*, 2nd ed. Addison-Wesley, 2003.

[80] Greenbaum, A. *Iterative Methods for Solving Linear Systems.* SIAM Publications, Philadelphia, PA, USA, 1997.

[81] Greenbaum, A., and Strakos, Z. Predicting the Behavior of Finite Precision Lanczos and Conjugate Gradient Computations. *SIAM J. Matrix Anal. Appl. 13*, 1 (1992), 121–137.

[82] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, 1999.

[83] Gropp, W., Lusk, E., and Thakur, R. *Using MPI-2: Advanced Features of the Message-Passing Interface.* The MIT Press, 1999.

[84] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and van de Geijn, R. A. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software 27*, 4 (December 2001), 422–455.

[85] Hackbusch, W. *Multi-Grid Methods and Applications.* Springer-Verlag, 1985.

[86] Hackbusch, W. *Iterative Solution of Large Sparse Systems of Equations.* Springer-Verlag, 1994.

[87] Hageman, L. A., and Young, D. M. *Applied Iterative Methods.* Academic Press, 1981.

[88] Hagemann, M., and Schenk, O. Weighted Matchings for Preconditioning Symmetric Indefinite Linear Systems. *SIAM J. Sci. Comput. 28*, 2 (2006), 403–420.

[89] Hendrickson, B., and Rothberg, E. Improving the Run Time and Quality of Nested Dissection Ordering. *SIAM J. Sci. Comput. 20*, 2 (1998), 468–489.

[90] Hennessy, J. L., and Patterson, D. *Computer Architecture. A Quantitative Approach*, 4 ed. Morgan Kaufmann Publishers, 2006.

[91] Henson, V. E., and Yang, U. M. BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Numer. Math. 41*, 1 (2002), 155–177.

[92] Heroux, M. A., Bartlett, R. A., Howle, V. E., Hoekstra, R. J., Hu, J. J., Kolda, T. G., Lehoucq, R. B., Long, K. R., Pawlowski, R. P., Phipps, E. T., Andrew G. Salinger, H. K. T., Tuminaro, R. S., James M. Willenbring, A. W., and Stanley, K. S. An overview of the Trilinos Project. *ACM Trans. Math. Software 3*, 33 (2005), 397—-423.

[93] Hestenes, M., and Stiefel, E. Methods of Conjugate Gradients for Solving Linear Systems. *J. Research Nat. Bur. Standards 49* (1952), 409–436.

[94] Higham, N. J. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[95] Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H., and Zahir, R. Introducing the IA-64 Architecture. *IEEE Micro 20*, 5 (2000), 12–23.

[96] Hysom, D., and Pothen, A. A Scalable Parallel Algorithm for Incomplete Factor Preconditioning. *SIAM J. Sci. Comput. 22*, 6 (2000), 2194–2215.

[97] Hysom, D., and Pothen, A. Level-based Incomplete LU Factorization: Graph Model and Algorithms. Tech. Rep. UCRL-JC-150789, Lawrence Livermore National Labs, November 2002.

[98] Intel. Intel Itanium 2 Processor Reference Manual - For Software Development and Optimization, May 2004. Document-No: 251110-003.

[99] JENNINGS, A., and MALIK, G. M. Partial Elimination. *IMA J. Appl. Math. 20*, 3 (1977), 307–316.

[100] Jones, M. T., and Plassmann, P. E. Scalable Iterative Solution of Sparse Linear Systems. *Parallel Comput. 20*, 5 (1994), 753–773.

[101] Karypis, G., and Kumar, V. Parallel Threshold-Based ILU Factorization. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)* (New York, NY, USA, 1997), ACM, pp. 1–24.

[102] Karypis, G., and Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[103] Karypis, G., and Kumar, V. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0*, September 1998.

[104] Karypis, G., and Kumar, V. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *J. Parallel and Distrib. Comput. 48*, 1 (1998), 71–95.

[105] Karypis, G., Schloegel, K., and Kumar, V. *ParMeTis: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1.* University of Minnesota, Dept. of Computer Science and Engineering, September 2003. `http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf`.

[106] Kernighan, B., and Lin, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal 29*, 2 (1970), 291–307.

[107] Kershaw, D. S. The Incomplete Cholesky–Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations. *Journal of Computational Physics 26*, 1 (1978), 43 – 65.

[108] Lanczos, C. Solution of Systems of Linear Equations by Minimized Iterations. *J. Research Nat. Bur. Standards 49*, 33 (1952), 33–53.

[109] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw. 5*, 3 (1979), 308–323.

[110] Li, N., Saad, Y., and Chow, E. Crout Versions of ILU for General Sparse Matrices. *SIAM J. Sci. Comput. 25*, 2 (2003), 716–728.

[111] Li, Z., Saad, Y., and Sosonkina, M. pARMS: a Parallel Version of the Algebraic Recursive Multilevel Solver. *Numer. Alg. Appl. 10* (2003), 485–509.

[112] Lipton, R. J., Rose, D. J., and Tarjan, R. E. Generalized Nested Dissection. *SIAM J. Numer. Anal. 16*, 2 (1979), 346–358.

[113] MacLachlan, S., and Saad, Y. A Greedy Strategy for Coarse-Grid Selection. *SIAM J. Sci. Comput. 29*, 5 (2007), 1825–1853.

[114] Manteuffel, T. A. An Incomplete Factorization Technique for Positive Definite Linear Systems. *Mathematics of Computation 34*, 150 (1980), 473–497.

[115] Mattson, T. G., Sanders, B. A., and Massingill, B. L. *Patterns for Parallel Programming*. Addison-Wesley Publishers, 2004.

[116] McCormick, S. F. *Multigrid Methods*. SIAM Books, Philadelphia, PA, USA, 1987.

[117] McNairy, C., and Soltis, D. Itanium 2 Processor Microarchitecture. *IEEE Micro 23*, 2 (2003), 44–55.

[118] Meijerink, J., and van der Vorst, H. An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-matrix. *Math. Comp. 137*, 31 (1977), 148–162.

[119] Meurant, G. *Computer Solution of Large Linear Systems*. North-Holland, 1999.

[120] monga Made, M. M., and van der Vorst, H. A. Parallel Incomplete Factorizations with pseudo-Overlapped Subdomains. *Parallel Comput. 27*, 8 (2001), 989–1008.

[121] Munksgaard, N. Solving Sparse Symmetric Sets of Linear Equations by Preconditioned Conjugate Gradients. *ACM Trans. Math. Softw. 6*, 2 (1980), 206–219.

[122] Notay, Y. Flexible Conjugate Gradients. *SIAM J. Sci. Comput. 22*, 4 (2000), 1444–1460.

[123] Olschowka, M., and Neumaier, A. A New Pivoting Strategy for Gaussian Elimination. *Linear Algebra Appl. 240* (1996), 131 – 151.

[124] Pachecho, P. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[125] Paige, C., and Saunders, M. A. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM J. Numer. Anal. 12* (1975), 617–629.

[126] pARMS: Parallel Algebraic Recursive Multilevel Solvers. `http://www-users.cs.umn.edu/saad/software/pARMS/`.

[127] Peaceman, D. W., and Rachford, H. H. The Numerical Solution of Parabolic and Elliptic Differential Equations. *Journal of the Society for Industrial and Applied Mathematics 3*, 1 (1955), 28–41.

[128] SCOTCH: static mapping, graph partitioning, and sparse matrix block ordering package. `http://www.labri.fr/perso/pelegrin/scotch/`.

[129] Pellegrini, F. *PT-SCOTCH and LIBSCOTCH User's Guide, version 5.0.6*. ScAlApplix project, INRIA Bordeaux Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, Université Bordeaux I, 2008.

[130] Pellegrini, F. *SCOTCH and LIBSCOTCH User's Guide, version 5.0.6*. ScAlApplix project, INRIA Bordeaux Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, Université Bordeaux I, 2008.

[131] Pellegrini, F., Roman, J., and Amestoy, P. Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing* (London, UK, 1999), vol. 1586/1999 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 986–995.

[132] Quarteroni, A. M., and Valli, A. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, 1999.

[133] Reid, J. On the Method of Conjugate Gradients for the Solution of Large Sparse Systems of Linear Equations. In *Large Sparse Sets of Linear Equations: Proceedings of the Oxford conference of the Institute of Mathematics and Its Applications* (1971), J. Reid, Ed., London: Academic Press, pp. 231–254.

[134] Ruge, J., and Stüben, K. Algebraic Multigrid. In *Multigrid Methods. Frontiers in Applied Mathematics* (1987), S. McCormick, Ed., vol. 3, SIAM Publications, pp. 73–130.

[135] Saad, Y. Preconditioning Techniques for Nonsymmetric and Indefinite Linear Systems. *J. Comput. Appl. Math. 24*, 1–2 (1988), 89–105.

[136] Saad, Y. A Flexible Inner-Outer Preconditioned GMRES Algorithm. *SIAM J. Sci. Comput. 14*, 2 (1993), 461–469.

[137] Saad, Y. ILUT: a Dual Threshold Incomplete ILU Factorization. *Numer. Lin. Alg. w. Appl. 1* (1994), 387–402.

[138] Saad, Y. *Iterative Methods for Sparse Linear Systems*, second ed. SIAM Publications, 2003.

[139] Saad, Y., and Schultz, M. H. GMRES: a Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Comput. 7* (1986), 856–869.

[140] Saad, Y., and Suchomel, B. ARMS: An Algebraic Recursive Multilevel Solver for General Sparse Linear Systems. *Numer. Lin. Alg. w. Appl. 9*, 5 (2002), 359–378.

[141] Schenk, O., Bollhöfer, M., and Römer, R. A. On Large Scale Diagonalization Techniques For the Anderson Model of Localization. *SIAM Review 50* (2008), 91–112.

[142] Schenk, O., and Gärtner, K. Two-Level Dynamic Scheduling in PARDISO: Improved Scalability on Shared-Memory Multiprocessing Systems. *Parallel Comput. 28*, 2 (2002), 187–197.

[143] Schenk, O., and Gärtner, K. On Fast Factorization Pivoting Methods for Sparse Symmetric Indefinite Systems. Technical Report CS-2004-005, Department of Computer Science, University of Basel, 2004. Submitted.

[144] Schenk, O., and Gärtner, K. Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. *J. of Future Generation Computer Systems 20*, 3 (2004), 475–487.

[145] Schenk, O., Gärtner, K., and Fichtner, W. Efficient Sparse LU Factorization with Left-Right Looking Strategy on Shared-Memory Multiprocessors. *BIT 40*, 1 (2000), 158–176.

[146] Schlansker, M. S., and Rau, B. R. EPIC: Explicitly Parallel Instruction Computing. *Computer 33*, 2 (2000), 37–45.

[147] Schöberl, J., Gerstmayr, H., and Gaisbauer, R. NETGEN - Automatic Mesh Generator. `http://www.hpfem.jku.at/netgen`.

[148] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. Larrabee: a Many-Core x86 Architecture for Visual Computing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), ACM, pp. 1–15.

[149] SIAM. Graduate Education for Computational Science and Engineering. `http://www.siam.org/students/resources/report.php`.

[150] Smith, B., Bjorstad, P. E., and Gropp, W. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.

[151] Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., and Packer, C. V. Beowulf: A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing* (1995), CRC Press, pp. 11–14.

[152] Strakos, Z., and Tichy, P. Error Estimation in Preconditioned Conjugate Gradients. *BIT Numerical Mathematics 45*, 4 (2005), 789–817.

[153] Strohmaier, E., Dongarra, J. J., Meuer, H. W., and Simon, H. D. The Marketplace of High-Performance Computing. *Parallel Comput. 25*, 13–14 (1999), 1517–1544.

[154] Strohmaier, E., Dongarra, J. J., Meuer, H. W., and Simon, H. D. Recent Trends in the Marketplace of High-Performance Computing. *Parallel Comput. 31*, 3–4 (2005), 261–273.

[155] Szyld, D. B., and Vogel, J. A. FQMR: a Flexible Quasi-Minimal Residual Method with Inexact Preconditioning. *SIAM J. Sci. Comput. 23* (2001), 363–380.

[156] Trottenberg, U., Oosterlee, C., and SchÄuller, A. *Multigrid*. Academic Press, 2001.

[157] Tuminaro, R. S., Heroux, M. A., Hutchinson, S. A., and Shadid, J. N. Official Aztec User's Guide - Version 2.1, 1999. `http://www.cs.sandia.gov/CRF/aztec1.html`.

[158] van der Vorst, H. A. Iterative Solution Methods for Certain Sparse Linear Systems with a Non-symmetric Matrix Arising from PDE-problems. *Journal of Computational Physics 44*, 1 (1981), 1 – 19.

[159] van der Vorst, H. A. BI-CGSTAB: a Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Statist. Comput. 13*, 2 (1992), 631–644.

[160] Vanek, P., Brezina, M., and Mandel, J. Convergence of Algebraic Multigrid Based on Smoothed Aggregation. *Numer. Math. 88* (2001), 559–579.

[161] Vanek, P., Mandel, J., and Brezina, M. Algebraic Multigrid based on Smoothed Aggregation for Second and Fourth Order Problems. *Computing 56* (1996), 179–196.

[162] Varga, R., Saff, E., and Mehrmann, V. Incomplete Factorizations of Matrices and Connections with H-matrices. *SIAM J. Numer. Anal. 17* (1980), 787–793.

[163] Varga, R. S. *Matrix Iterative Analysis*. Prentice-Hall, 1962.

[164] Watkins, D. S. *Fundamentals of Matrix Computations*, 2nd ed. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[165] Watts, J. W. A Conjugate Gradient Truncated Direct Method for the Iterative Solution of the Reservoir Simulation Pressure Equation. *Society of Petrol Engineer Journal 21*, 3 (1981), 345–353.

[166] Weiss, R. *Parameter-Free Iterative Linear Solvers*. Akademie Verlag, 1996.

[167] Yang, U. M. *Algebraic Multigrid Methods — High Performance Preconditioners*, vol. 51 of *Lecture Notes in Computational Science and Engineering*. Springer, 2006, pp. 209–236.

[168] Young, D. M. *Iterative Methods for Solving Partial Difference Equations of Elliptic Type*. PhD thesis, Harvard University, Cambridge, Mass, 1950.

[169] Young, D. M. *Iterative Solution of Large Linear Systems*. Academic Press, 1971.

Apéndice A

Almacenamiento de matrices dispersas y operaciones matriciales básicas

En este apéndice presentamos algunos formatos/esquemas para el almacenamiento de matrices dispersas. Además, discutimos brevemente la implementación del producto de una matriz dispersa por un vector, una de las operaciones matriciales básicas que surge con mayor frecuencia en los métodos numéricos de resolución de problemas dispersos. Matricialmente, esta operación tiene la forma $y \leftarrow Ax$, donde $A \in \mathbb{R}^{n,n}$ es una matriz dispersa, y x e $y \in \mathbb{R}^n$ son vectores. Esta operación también se puede definir para matrices rectangulares, aunque hemos escogido este caso ya que la matriz de coeficientes del sistema lineal es cuadrada. El coste en número de operaciones aritméticas de esta operación es, en todos los formatos de almacenamiento presentados en este apéndice, $2 \cdot nnz(A)$, frente a las $2n^2$ que requiere el producto de una matriz por un vector cuando ésta presenta una estructura densa.

El formato más simple para almacenar una matriz dispersa es el coordinado (*COOrdinate format*, COO). Por cada entrada de la matriz, este formato almacena un par de números enteros con su índice de fila y su índice de columna, y un número real con el valor de la entrada. Por lo tanto, se necesitan tres vectores para almacenar la matriz, cada uno de dimensión $nnz(A)$. Consideremos por ejemplo la matriz

$$A = \begin{pmatrix} 1.1 & 0 & 2.2 & 0 & 0 \\ 0 & 3.3 & 4.4 & 0 & 0 \\ 0 & 0 & -5.5 & -6.6 & 0 \\ 7.7 & -8.8 & 0 & 9.9 & 0 \\ 0 & 0 & 0 & 0 & 10.1 \end{pmatrix}, \tag{A.1}$$

con $n = 5$ y $nnz(A) = 10$. En el formato COO, esta matriz se almacena en forma de tres vectores:

$$\begin{aligned} \texttt{AA} &= \{\ 1.1,\ 2.2,\ 3.3,\ 4.4,\ -5.5,\ -6.6,\ 7.7,\ -8.8,\ 9.9,\ 10.1\ \}, \\ \texttt{IR} &= \{\ 1,\ 1,\ 2,\ 2,\ 3,\ 3,\ 4,\ 4,\ 4,\ 5\ \}, \\ \texttt{IC} &= \{\ 1,\ 3,\ 2,\ 3,\ 3,\ 4,\ 1,\ 2,\ 4,\ 5\ \}. \end{aligned}$$

El vector `AA` contiene los valores de las entradas, y los vectores de números enteros `IR` e `IC` almacenan, respectivamente, los índices de fila y de columna de las entradas de A. Por lo tanto, el coste de almacenamiento del formato COO es $3 \cdot nnz(A)$. Desde el punto de vista del consumo de memoria, el formato COO no es óptimo; en cambio, presenta una gran flexibilidad a la hora de insertar o eliminar nuevas entradas, puesto que las entradas se pueden almacenar en cualquier orden. El Algoritmo A.1 calcula el producto de una matriz dispersa almacenada según el formato COO por un vector. El acceso a los vectores x e y involucra un direccionamiento *indirecto*, lo que supone un aumento del tiempo de acceso en comparación con el direccionamiento directo característico de los códigos equivalentes para matrices densas.

Algoritmo A.1: Calcula $y \leftarrow Ax$, donde A se almacena según el formato COO.

```
1 y ← 0
2 para k = 1 : nnz(A) hacer
3     y_IR[k] ← y_IR[k] + AA[k] · x_IC[k]
4 fin para
```

El esquema de almacenamiento probablemente más popular es el comprimido por filas (*Compress Storage Row*, CSR). Este formato reduce el consumo de memoria evitando la redundancia presente en los vectores de índices del formato COO. Los valores de las entradas de A se almacenan, ordenados por filas, en un vector números reales, `AA`, y sus índices de columna, en un vector de números enteros, `JA`; es decir, el valor `AA[j]` se sitúa en la columna `JA[j]` de la matriz A. El tamaño de ambos vectores es $nnz(A)$. Un segundo vector de enteros, `IA`, de tamaño $n+1$, contiene, en su posición k-ésima, $k = 1, 2, \ldots, n$, un índice al comienzo de la fila k-ésima de A, y en la posición $(n+1)$-ésima, el valor `IA[1]` $+ nnz(A)$, que se corresponde con el comienzo de la fila "simbólica" $(n+1)$-ésima. Así, los valores de la fila k-ésima de A se almacenan en el rango de posiciones `AA[IA[k]]`, `AA[IA[k]+1]`,..., `AA[IA[k+1]-1]` y, sus índices de columna, en el rango `JA[IA[k]]`, `JA[IA[k]+1]`,..., `JA[IA[k+1]-1]`. Para el caso de la matriz (A.1), estos tres vectores almacenarían los siguientes valores:

```
AA = {  1.1,  2.2,  3.3,  4.4, -5.5, -6.6,  7.7, -8.8,  9.9,  10.1  },
JA = {    1,    3,    2,    3,    3,    4,    1,    2,    4,     5  },
IA = {    1,    3,    5,    7,  10 ,   11    }.
```

En comparación con el formato COO, el coste de almacenamiento se reduce a $2 \cdot nnz(A) + n + 1$. La implementación del producto matriz-vector disperso se muestra en el Algoritmo A.2 y, como se puede comprobar, también realiza un direccionamiento indirecto en el acceso al vector x.

Algoritmo A.2: Calcula $y \leftarrow Ax$, donde A se almacena según el formato CSR.

```
1 y ← 0
2 para i = 1 : n hacer
3     para j = IA[i] : IA[i + 1] − 1 hacer
4         y_i ← y_i + AA[j] · x_JA[j]
5     fin para
6 fin para
```

El formato CSR presenta multitud de variantes y extensiones. La más común es aquella en la que A se almacena por columnas, dando lugar al formato comprimido por columnas (*Compress Storage Column*, CSC). Otro ejemplo es el formato comprimido por filas modificado (*Modified compress Storage Row*, MSR), que resulta ventajoso si un método numérico accede con mucha frecuencia a los elementos de la diagonal principal de A. En concreto, este formato almacena los elementos de la diagonal de A en las primeras n posiciones del vector `AA`, facilitando un rápido acceso. La posición $n+1$-ésima de este vector está vacía y, las posiciones posteriores, almacenan, por filas, los valores de las entradas que no pertenecen a la diagonal principal. Además, los vectores `IA` y `JA` del formato CSR se fusionan en un solo vector, `JA`, que contiene, en sus primeras $n+1$ posiciones, los punteros al comienzo de cada fila de A y, en las posiciones posteriores, los índices de columna de las entradas de A que no pertenecen a la diagonal principal. Para el caso de la matriz (A.1), los vectores `AA` y `JA` almacenarían los siguientes valores:

```
AA = {  1.1,  3.3, -5.5,  9.9,  10.1,  ∗,  2.2,  4.4, -6.6,  7.7, -8.8  },
JA = {    7,    8,    9,  10 ,  12  ,12 ,    3,    3,    4,    1,    2  }.
```

La implementación del producto matriz-vector disperso para el formato MSR sólo cambia ligeramente con respecto a la del CSR, como se muestra en el Algoritmo A.3.

Algoritmo A.3: Calcula $y \leftarrow Ax$, donde A se almacena según el formato MSR.

```
1 y ← 0
2 para i = 1 : n hacer
3     y_i ← AA[i] · x_i
4     para j = JA[i] : JA[i + 1] − 1 hacer
5         y_i ← y_i + AA[j] · x_JA[j]
6     fin para
7 fin para
```

Además de los formatos presentados, que son los más difundidos en la actualidad, existen multitud de esquemas y técnicas alternativas para el almacenamiento de matrices dispersas. Las particularidades de la aplicación en la que surge la matriz, su estructura concreta, los métodos numéricos que operan con la matriz o la arquitectura del computador que se emplee para resolver el problema, son los factores que determinan finalmente la forma más eficiente de almacenar una matriz dispersa. En [56, 62, 138] se ofrecen más detalles y algunos formatos de almacenamiento adicionales, así como una descripción de la implementación eficiente de otras operaciones matriciales básicas para matrices dispersas.

Apéndice B

El Multiprocesador SGI Altix 350

Los experimentos de la tesis se han llevado a cabo en un multiprocesador SGI Altix 350. La arquitectura de esta familia de multiprocesadores implementa un espacio de direcciones *compartido* por múltiples microprocesadores Itanium 2 (versión *Madison*) de Intel. El mantenimiento de la coherencia en el sistema de memoria es responsabilidad del hardware del multiprocesador y, además, el coste de acceso a memoria principal depende de la posición a la que se accede (es decir, es más rápido acceder a ciertas posiciones que a otras). Por tanto, los multiprocesadores de la familia SGI Altix son del tipo ccNUMA (*cache-coherent Non-Uniform Memory Access*) [79]. En este apéndice describimos el microprocesador Itanium 2, los bloques básicos de los multiprocesadores de la familia SGI Altix 350, y la configuración concreta del multiprocesador en el que se han realizado los experimentos de la tesis.

B.1. El microprocesador Itanium 2

El procesador Itanium 2 es la segunda implementación de la arquitectura Itanium (denominada formalmente IA-64 [95]), mejorada gracias al aumento de la frecuencia de reloj, y a ciertas modificaciones en la profundidad y control del cauce segmentado, el esquema de predicción de saltos y en la interfaz del sistema de memoria [117]. En su versión *Madison*, el Itanium 2 se implementa con tecnología de 130 nm, en un circuito integrado con una superficie de 374 mm^2 que contiene 410 millones de transistores. El procesador funciona a una frecuencia de 1,5 GHz, con una microarquitectura segmentada de 8 etapas, de ejecución en orden, capaz de ejecutar un máximo de 6 instrucciones por cada ciclo de reloj. La microarquitectura dispone de 11 unidades funcionales, dos para el procesamiento de enteros (unidades I), cuatro para operaciones de acceso a memoria (unidades M), tres para procesamiento de saltos (unidades B) y dos para operaciones de coma flotante (unidades F). El Itanium 2 *Madison* dispone de tres niveles de caché dispuestos del modo ilustrado en la Figura B.1. La caché L1 está separada en dos cachés de 16 KBytes, una para datos, denominada L1D, y otra para instrucciones, L1I. Los niveles L2 y L3 son cachés unificadas de tamaño 256 KBytes y 6 MBytes, respectivamente. La interfaz del sistema funciona a 200 MHz con un bus de 128 bits (16 bytes) de datos que opera síncronamente a un ritmo de 400 millones de transferencias por segundo (16 bytes $\times$ 400 Megatransferencias/seg. = 6,4 GBytes/seg.). El procesador también incluye soporte en hardware para acelerar el proceso de traducción de direcciones virtuales a direcciones físicas. En concreto, incorpora dos tablas de traducción rápida de direcciones (*Translation Look-ahead Buffer*, TLB) con dos niveles cada una, una para instrucciones (TLB L1I y L2I) y otra para datos (TLB L1D y L2D), junto con un mecanismo hardware denominado HPW [98] (*Hardware Page Walking*) para reducir el número de invocaciones al sistema operativo.

El procesador Itanium 2 implementa la arquitectura IA-64, una metodología avanzada para el aprovechamiento del paralelismo a nivel de instrucción (*instruction-level parallelism*, ILP) denominada EPIC [146] (*Explicit Parallel Instruction Computation*). EPIC se basa en la premisa

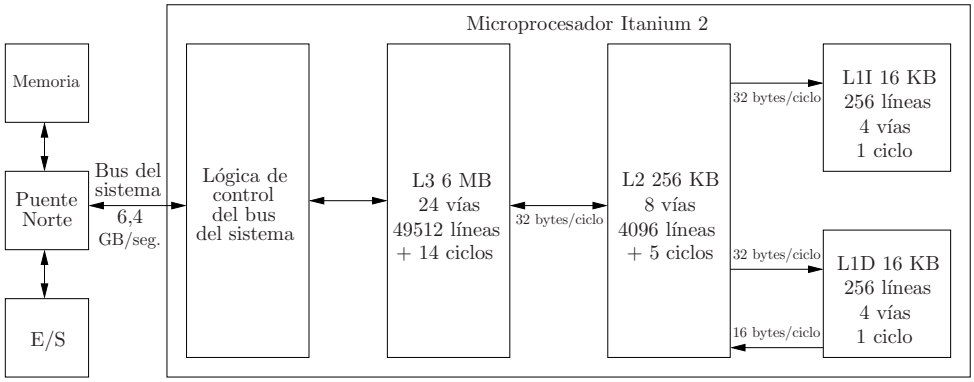


Figura B.1: Jerarquía de caché del procesador Itanium 2 [98]. La figura muestra, para cada memoria caché, de izquierda a derecha y de abajo a arriba, su nivel y tipo (I para instrucciones, D para datos, sin letra para caché unificada), tamaño, grado de asociatividad, número de líneas y latencia en ciclos de reloj.

de que el compilador dispone de una mejor visibilidad del programa que la unidad del control del procesador, aprovechando el ILP a través de: (1) mecanismos para que el compilador reorganice el código conforme al conocimiento adquirido del mismo; (2) registros y unidades funcionales suficientes para realizar operaciones en paralelo y guardar resultados intermedios; y (3) formatos de instrucciones para que el compilador comunique al hardware información clave acerca del programa compilado.

En la arquitectura IA-64 aparecen dos conceptos ligados al aprovechamiento del ILP y a facilitar la decodificación. Por una parte, están los haces (del término inglés *bundle*) de instrucciones y, por otra, los grupos de instrucciones. Los haces de instrucciones tienen un tamaño de 128 bits y codifican 3 instrucciones cada uno. Cada una de estas instrucciones se codifica dentro del haz mediante un campo de 41 bits, de los cuales 14 se utilizan para el código de operación, 7 para codificar el identificador de cada uno de los registros que se usan como operandos fuente y resultado, y 6 para codificar el identificador de un registro del tipo predicado; la finalidad de los 5 bits restantes del haz se expone en el siguiente párrafo, junto con el concepto de grupo de instrucciones. El repertorio de instrucciones incluye operaciones aritmético-lógicas con enteros, operaciones con enteros no aritmético-lógicas, operaciones de acceso a memoria, operaciones aritméticas de coma flotante, instrucciones de salto e instrucciones especiales extendidas. Las instrucciones tienen acceso a 64 registros de un 1 bit para almacenar predicados, 128 registros de propósito general de 64 bits para almacenar números enteros, 128 registros de 82 bits para almacenar números en coma flotante, y 8 registros de 64 bits para saltos indirectos. Además, también existen otros 128 registros de aplicación o propósito específico de 64 bits que sirven, por ejemplo, para facilitar al compilador la implementación de la segmentación software con la ayuda de la lógica de renombramiento de la que dispone el procesador [117].

Además del concepto de haz, en la arquitectura IA-64 aparece el concepto de grupo de instrucciones. Un grupo de instrucciones está constituido por un número arbitrario de instrucciones que se pueden ejecutar simultáneamente en las unidades funcionales de la microarquitectura. Durante la compilación, se analiza estáticamente el programa para identificar grupos de instrucciones y, posteriormente, se planifica su ejecución distribuyendo las instrucciones entre los haces, que constituyen finalmente el código que procesa la microarquitectura. El compilador no puede incluir cualquier combinación de operaciones dentro de un mismo haz de instrucciones (por ejemplo, no es posible agrupar tres operaciones del tipo M en un mismo haz). Sin embargo, a diferencia de otras arquitecturas VLIW, los haces de instrucciones sí pueden agrupar instrucciones pertenecientes a grupos distintos (es decir, instrucciones que pueden ser dependientes entre sí). El compilador debe indicar explícitamente los límites de los grupos situando indicadores (*stops*)

entre las instrucciones que pertenecen a grupos distintos. Esta información se incluye en los 5 bits restantes del código del haz, denominados bits de marca, junto con las unidades funcionales en las que se van a ejecutar las instrucciones del haz. Gracias al uso de los bits de marca se consiguen códigos VLIW más compactos, además de facilitarse la decodificación y aumentar la flexibilidad de uso de las unidades funcionales disponibles. Cabe destacar que los bits de marca son uno de los recursos a través del cual el compilador se comunica con la arquitectura.

La Figura B.2 ilustra el cauce segmentado del Itanium 2. Las dos primeras etapas (IPG y ROT) se encargan de captar las instrucciones y constituyen la denominada *antesala* del procesador (*front-end*); el resto de etapas constituyen el *núcleo* del procesador (*back-end*) y se encargan del procesamiento de los haces de instrucciones.

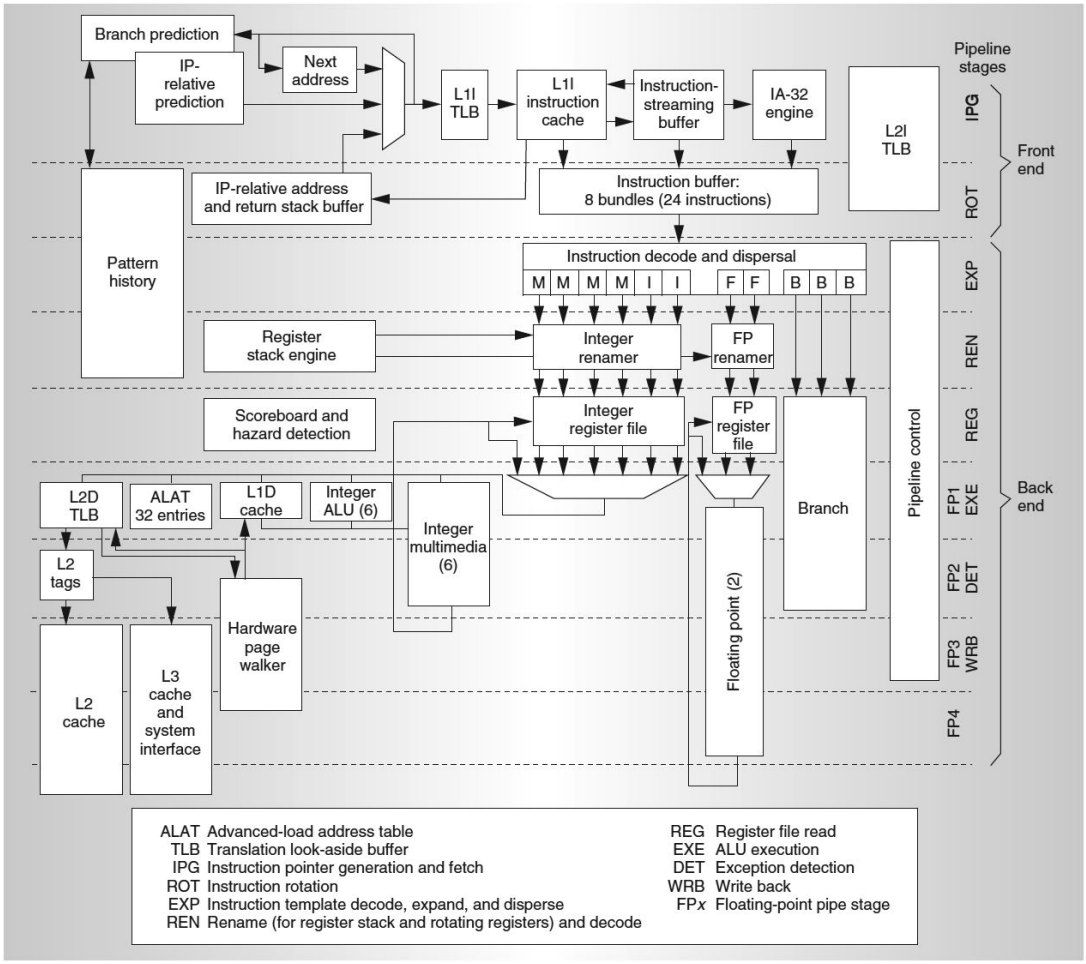


Figura B.2: Cauce segmentado del microprocesador Itanium 2 [117].

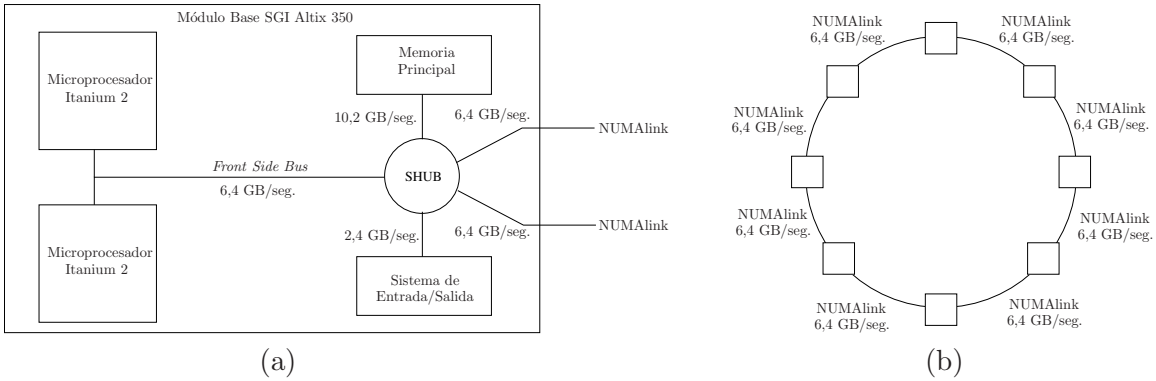
La etapa IPG determina la dirección virtual del haz que se va a captar a continuación, traduce su dirección virtual a dirección física mediante la TLB L1I, y capta este haz de la caché L1I. Esta dirección virtual puede corresponderse con la dirección del siguiente haz de instrucciones en el código máquina, o bien con los punteros generados por la lógica de predicción de saltos, de recuperación de predicción errónea, o de procesamiento de excepciones. La latencia de acceso a la caché L1I es de un ciclo, y ésta puede proporcionar 32 bytes (es decir, 2 haces) por cada ciclo. Los haces captados por la etapa IPG se almacenan en un buffer de instrucciones gestionado por la etapa ROT. Este buffer tiene espacio para 8 haces (24 instrucciones) y presenta dos haces de instrucciones (6 instrucciones) por cada ciclo a la etapa EXP, que se encarga de la decodificación de los bits de marca de los haces y de la distribución de las instrucciones entre las 11 las unidades funcionales conforme a dichos bits. Aunque la etapa EXP intenta

asignar el máximo número de instrucciones a los recursos disponibles, puede darse el caso de que ésta no logre asignar por completo alguno de los dos haces. La etapa EXP devuelve al buffer de instrucciones los haces que no han podido ser asignados a las unidades (cero, uno o dos), junto con el número de haces que han sido procesados (dos, uno o cero). La etapa REN asigna registros virtuales a registros físicos del banco de registros y decodifica las instrucciones. La etapa REG accede a estos registros y proporciona los operandos a las unidades funcionales, y también detecta y gestiona los riesgos de datos entre instrucciones, parando la emisión de las instrucciones si fuese necesario. La etapa EXE/FP1 ejecuta las operaciones con enteros o comienza la ejecución segmentada de una operación de coma flotante o multimedia. Todas las operaciones con enteros tienen una latencia de un ciclo de reloj y utilizan caminos de *bypass* para permitir que el resultado generado por una de ellas esté disponible a la entrada de cualquier otra unidad funcional tan pronto como dicho resultado se haya obtenido. Las operaciones de coma flotante tienen una latencia de cuatro ciclos, y las unidades funcionales que las implementan están completamente segmentadas y utilizan caminos de *bypass*. Además de las operaciones de suma, resta, multiplicación y división, estas unidades también implementan operaciones de multiplicación-acumulación. Cuando el compilador puede incluir una operación de este último tipo en cada haz de instrucciones, el procesador puede realizar hasta un máximo de 4 operaciones aritméticas en coma flotante (*flops*) por ciclo y, por tanto, la velocidad pico del Itanium 2 *Madison* es de 6 Gigaflops/seg. (1,5 GHz/seg. $\times$ 4 flops/ciclo). La etapa DET es la última en la que se pueden detectar excepciones. La validación de la predicción de la dirección destino de salto se realiza también en esta etapa. Si la predicción de la condición del salto o la dirección destino no son correctas, esta fase se encarga de proporcionar la dirección correcta a la etapa IPG. Por último, la etapa WRB escribe los resultados de las operaciones en el banco de registros. Es posible consultar una explicación más detallada del cauce en [117].

B.2. Configuración del multiprocesador SGI Altix 350

Aunque los multiprocesadores SGI Altix 350 se construyen a partir de módulos base, extensión y enrutador, el sistema utilizado en la fase experimental de la tesis únicamente dispone de módulos base, tal y como se ilustra en la Figura B.3 (a). Cada módulo base dispone de 2 microprocesadores Itanium 2, y una porción de tamaño 4 GBytes de la memoria global del multiprocesador. Los módulos se combinan para configurar sistemas más grandes usando las dos conexiones bidireccionales NUMAlink 4 disponibles en cada módulo base. Cada conexión permite enviar y recibir datos simultáneamente, con un ancho de banda pico de 3,2 GBytes/seg. por cada sentido (6,4 GBytes/seg. en total). Los módulos conectados forman un multiprocesador con memoria compartida del tipo ccNUMA, de tal forma que un único proceso puede reservar toda la memoria disponible en el sistema. La Figura B.3 (b) muestra la topología de la red de interconexión del sistema, formado por 8 módulos base, es decir, 16 procesadores Itanium 2, conectados en anillo para compartir un total de 32 GBytes de memoria. Aunque dentro de un mismo módulo, dos procesadores comparten bus (*Front Side Bus*, FSB) y memoria, la memoria global del multiprocesador está *físicamente* distribuida entre los distintos módulos, aumentando de esta forma la escalabilidad del sistema.

El mantenimiento de la coherencia de memoria corre a cargo del hardware del multiprocesador. El SGI Altix 350 implementa un protocolo de coherencia basado en directorio distribuido entre los módulos base, utilizando los SHUBs y las conexiones NUMAlink para transmitir los paquetes generados por el protocolo de control de coherencia. Cuando el sistema se inicia, los SHUBs reservan una porción de la memoria local para almacenar, por cada bloque de esta memoria, su estado (por ejemplo, local, compartido o exclusivo), junto con una máscara de bits por cada uno de los procesadores del sistema. Estos bits se utilizan para identificar los procesadores que tienen una copia del bloque de memoria en su memoria caché. El SHUB también espía (*snoop*) la información de control de coherencia generada por el procesador Itanium 2 a través



(a) (b)

Figura B.3: El multiprocesador SGI Altix 350. (a) Módulo base. (b) Topología de la red de interconexión.

del FSB, y gestiona el directorio de forma acorde a esta información y a las peticiones recibidas desde nodos remotos.